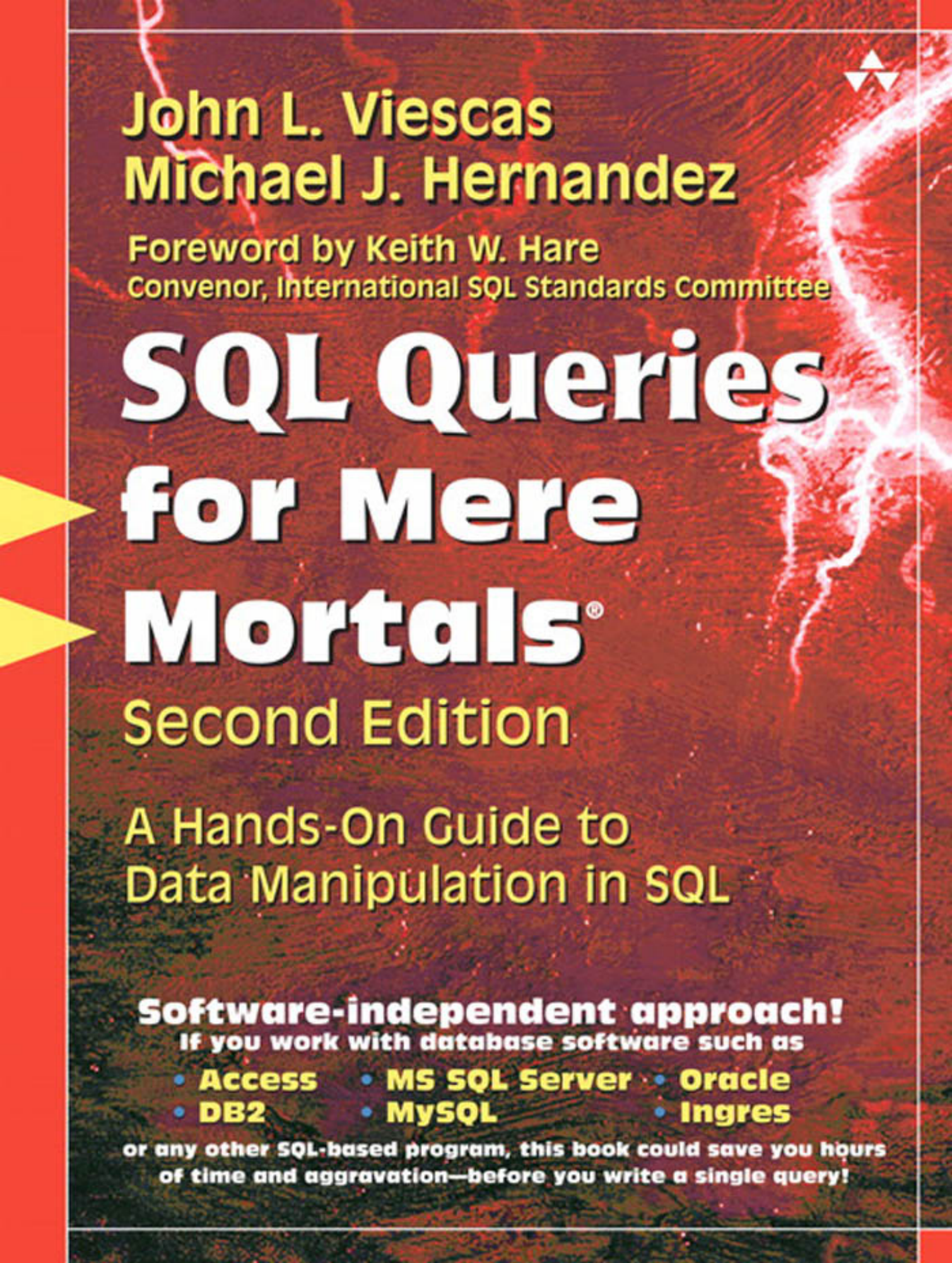
VISIT…


John L. Viescas
Michael J. Hernandez
Foreword by Keith W. Hare
Convenor, International SQL Standards Committee
SQL Queries for Mere Mortals®
Second Edition
A Hands-On Guide to Data Manipulation in SQL
Software-independent approach!
If you work with database software such as
• Access • MS SQL Server • Oracle
• DB2 • MySQL • Ingres
or any other SQL-based program, this book could save you hours of time and aggravation—before you write a single query!

Praise for *SQL Queries for Mere Mortals®, Second Edition*

Unless you are working at a very advanced level, this is the only SQL book you will ever need. The authors have taken the mystery out of complex queries and explained principles and techniques with such clarity that a "Mere Mortal" will indeed be empowered to perform the superhuman. Do not walk past this book!

— Graham Mandeno, Database Consultant

I learned SQL primarily from the first edition of this book, and I am pleased to see a second edition of this book so that others can continue to benefit from its organized presentation of the language. Starting from how to design your tables so that SQL can be effective (a common problem for database beginners), and then continuing through the various aspects of SQL construction and capabilities, the reader can become a moderate expert upon completing the book and its samples. Learning how to convert a question in English into a meaningful SQL statement will greatly facilitate your mastery of the language. Numerous examples from real life will help you visualize how to use SQL to answer the questions about the data in your database. Just one of the "watch out for this trap" items will save you more than the cost of the book when you avoid that problem when writing your queries. I highly recommend this book if you want to tap the full potential of your database.

— Kenneth D. Snell, Ph.D., Database Designer/Programmer

I don't think they do this in public schools any more, and it is a shame, but do you remember in the seventh and eighth grades when you learned to diagram a sentence? Those of you who do may no longer remember how you did it, but all of you do write better sentences because of it. John Viescas and Mike Hernandez must have remembered because they take everyday English queries and literally translate them into SQL. This is an important book for all database designers. It takes the complexity of mathematical Set Theory and of First Order Predicate Logic, as outlined in E. F. Codd's original treatise on relational database design, and makes it easy for anyone to understand. If you want an elementary- through intermediate-level course on SQL, this is the one book that is a requirement, no matter how many others you buy.

— Arvin Meyer, MCP, MVP

SQL Queries for Mere Mortals, Second Edition, provides a step-by-step, easy-to-read introduction to writing SQL queries. It includes hundreds of examples with detailed explanations. This book provides the tools you need to understand, modify, and create SQL queries.

— Keith W. Hare, Convenor, ISO/IEC JTC1 SC32 WG3—
the International SQL Standards Committee

Even in this day of wizards and code generators, successful database developers still require a sound knowledge of Structured Query Language (SQL, the standard language for communicating with most database systems). In this book, John and Mike do a marvelous job of making what's usually a dry and difficult subject come alive, presenting the material with humor in a logical manner, with plenty of relevant examples. I would say that this book should feature prominently in the collection on the bookshelf of all serious developers, except that I'm sure it'll get so much use that it won't spend much time on the shelf!

— Doug Steele, Microsoft Access Developer and author

SQL Queries *for* Mere Mortals®
Second Edition

This page intentionally left blank

SQL Queries *for* Mere Mortals® Second Edition

A Hands-On Guide to Data Manipulation in SQL

John L. Viescas
Michael J. Hernandez

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco •
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact: U.S. Corporate and Government Sales, (800) 382-3419 corpsales@pearsontechgroup.com

For sales outside the United States please contact: International Sales, international@pearsoned.com

Visit us on the Web: www.awprofessional.com

Library of Congress Cataloging-in-Publication Data
Viescas, John L., 1947-
SQL queries for mere mortals : a hands-on guide to data manipulation in SQL / John L. Viescas and Michael J. Hernandez. — 2nd ed.
p. cm.
On t.p. of previous ed. Michael J. Hernandez's name appeared first.
Includes index.
ISBN 0-321-44443-4 (pbk. : alk. paper)
1. SQL (Computer program language) 2. Database searching. I. Hernandez, Michael J. (Michael James), 1955- II. Viescas, John L., 1947- SQL queries for mere mortals. III. Title.

QA76.73.S67H48 2007
005.75'85—dc22 2007026881

Copyright © 2008 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, write to: Pearson Education, Inc., Rights and Contracts Department, 501 Boylston Street, Suite 900, Boston, MA 02116, Fax (617) 671 3447

ISBN-13: 978-0-321-44443-1
ISBN-10: 0-321-44443-4

Text printed in the United States on recycled paper at Courier in Stoughton, Massachusetts.
First printing, September 2007

Editor-in-Chief: Karen Gettman
Acquisitions Editor: Chuck Toporek
Managing Editor: John Fuller
Project Editor: Elizabeth Ryan
Copy Editor: Chrysta Meadowbrooke
Indexer: Coughlin Indexing
Proofreader: Mike Shelton
Technical Reviewers: Keith Hare, Stephen Forte
Cover Designer: Alan Clements
Composition: Pine Tree Composition

Contents

Foreword xvii

Preface xix

About the Authors xxi

Introduction xxiii

Are You a Mere Mortal? xxiii
About This Book xxiv
What This Book Is Not xxv
How to Use This Book xxvi
Reading the Diagrams Used in This Book xxvii
Sample Databases Used in This Book xxx
"Follow the Yellow Brick Road" xxxii

PART I Relational Databases and SQL 1

CHAPTER 1 What Is Relational? 3

Topics Covered in This Chapter 3
Types of Databases 3
A Brief History of the Relational Model 4
In the Beginning . . . 4
Relational Database Software 5
Anatomy of a Relational Database 6
Tables 6
Fields 7
Records 8
Keys 8

Views 9
Relationships 11
What's in It for You? 15
Where Do You Go from Here? 16
Summary 17

CHAPTER 2 Ensuring Your Database Structure Is Sound 19

Topics Covered in This Chapter 19
Why Is This Chapter Here? 19
Why Worry about Sound Structures? 20
Fine-Tuning Fields 21
What's in a Name? (Part One) 21
Smoothing Out the Rough Edges 23
Resolving Multipart Fields 25
Resolving Multivalued Fields 27
Fine-Tuning Tables 30
What's in a Name? (Part Two) 30
Ensuring a Sound Structure 32
Resolving Unnecessary Duplicate Fields 33
Identification Is the Key 39
Establishing Solid Relationships 42
Establishing a Deletion Rule 44
Setting the Type of Participation 46
Setting the Degree of Participation 48
Is That All? 50
Summary 51

CHAPTER 3 A Concise History of SQL 53

Topics Covered in This Chapter 53
The Origins of SQL 54
Early Vendor Implementations 55
". . . And Then There Was a Standard" 56
Evolution of the ANSI/ISO Standard 58
Other SQL Standards 61
Commercial Implementations 64
What the Future Holds 65
Why Should You Learn SQL? 65
Summary 66

PART II SQL Basics 69

CHAPTER 4 Creating a Simple Query 71

Topics Covered in This Chapter 71
Introducing SELECT 72
The SELECT Statement 73
A Quick Aside: Data versus Information 75
Translating Your Request into SQL 77
Expanding the Field of Vision 81
Using a Shortcut to Request All Columns 83
Eliminating Duplicate Rows 84
Sorting Information 87
First Things First: Collating Sequences 88
Let's Now Come to Order 89
Saving Your Work 92
Sample Statements 93
Summary 102
Problems for You to Solve 103

CHAPTER 5 Getting More Than Simple Columns 105

Topics Covered in This Chapter 105
What Is an Expression? 106
What Type of Data Are You Trying to Express? 107
Changing Data Types: The CAST Function 110
Specifying Explicit Values 112
Character String Literals 112
Numeric Literals 114
Datetime Literals 115
Types of Expressions 117
Concatenation 117
Mathematical Expressions 121
Date and Time Arithmetic 124
Using Expressions in a SELECT Clause 128
Working with a Concatenation Expression 128
Naming the Expression 129
Working with a Mathematical Expression 131

Working with a Date Expression 132
A Brief Digression: Value Expressions 133
That "Nothing" Value: Null 135
Introducing Null 136
The Problem with Nulls 138
Sample Statements 139
Summary 147
Problems for You to Solve 149

CHAPTER 6 Filtering Your Data 151

Topics Covered in This Chapter 151
Refining What You See Using WHERE 151
The WHERE Clause 152
Using a WHERE Clause 154
Defining Search Conditions 156
Comparison 156
Range 164
Set Membership 167
Pattern Match 169
Null 173
Excluding Rows with NOT 175
Using Multiple Conditions 178
Introducing AND and OR 179
Excluding Rows: Take Two 184
Order of Precedence 187
Checking for Overlapping Ranges 191
Nulls Revisited: A Cautionary Note 193
Expressing Conditions in Different Ways 197
Sample Statements 198
Summary 206
Problems for You to Solve 207

PART III Working with Multiple Tables 211

CHAPTER 7 Thinking in Sets 213

Topics Covered in This Chapter 213
What Is a Set, Anyway? 214

Operations on Sets 215
Intersection 216
Intersection in Set Theory 216
Intersection between Result Sets 217
Problems You Can Solve with an Intersection 221
Difference 222
Difference in Set Theory 222
Difference between Result Sets 224
Problems You Can Solve with Difference 227
Union 228
Union in Set Theory 228
Combining Result Sets Using a Union 230
Problems You Can Solve with Union 232
SQL Set Operations 233
Classic Set Operations versus SQL 233
Finding Common Values: INTERSECT 234
Finding Missing Values: EXCEPT (Difference) 236
Combining Sets: UNION 239
Summary 242

CHAPTER 8 INNER JOINs 243

Topics Covered in This Chapter 243
What Is a JOIN? 243
The INNER JOIN 244
What's "Legal" to JOIN? 244
Column References 245
Syntax 246
Check Those Relationships! 261
Uses for INNER JOINs 262
Find Related Rows 262
Find Matching Values 263
Sample Statements 263
Two Tables 264
More Than Two Tables 270
Looking for Matching Values 277
Summary 288
Problems for You to Solve 289

CHAPTER 9 OUTER JOINs 293

Topics Covered in This Chapter 293
What Is an OUTER JOIN? 293
The LEFT/RIGHT OUTER JOIN 295
Syntax 296
The FULL OUTER JOIN 314
Syntax 314
FULL OUTER JOIN on Non-Key Values 317
UNION JOIN 317
Uses for OUTER JOINs 318
Find Missing Values 318
Find Partially Matched Information 319
Sample Statements 319
Summary 335
Problems for You to Solve 335

CHAPTER 10 UNIONs 339

Topics Covered in This Chapter 339
What Is a UNION? 339
Writing Requests with UNION 342
Using Simple SELECT Statements 342
Combining Complex SELECT Statements 345
Using UNION More Than Once 349
Sorting a UNION 351
Uses for UNION 352
Sample Statements 353
Summary 365
Problems for You to Solve 366

CHAPTER 11 Subqueries 369

Topics Covered in This Chapter 369
What Is a Subquery? 370
Row Subqueries 370
Table Subqueries 371
Scalar Subqueries 372

Subqueries as Column Expressions 372
Syntax 372
An Introduction to Aggregate Functions: COUNT and MAX 375
Subqueries as Filters 377
Syntax 378
Special Predicate Keywords for Subqueries 379
Uses for Subqueries 392
Build Subqueries as Column Expressions 392
Use Subqueries as Filters 393
Sample Statements 394
Subqueries in Expressions 395
Subqueries in Filters 400
Summary 409
Problems for You to Solve 410

PART IV Summarizing and Grouping Data 413

CHAPTER 12 Simple Totals 415

Topics Covered in This Chapter 415
Aggregate Functions 416
Counting Rows and Values with COUNT 418
Computing a Total with SUM 421
Calculating a Mean Value with AVG 423
Finding the Largest Value with MAX 424
Finding the Smallest Value with MIN 426
Using More Than One Function 427
Using Aggregate Functions in Filters 428
Sample Statements 431
Summary 438
Problems for You to Solve 439

CHAPTER 13 Grouping Data 441

Topics Covered in This Chapter 441
Why Group Data? 442
The GROUP BY Clause 444
Syntax 445
Mixing Columns and Expressions 450

Using GROUP BY in a Subquery in a WHERE Clause 452
Simulating a SELECT DISTINCT Statement 453
"Some Restrictions Apply" 454
Column Restrictions 455
Grouping on Expressions 457
Uses for GROUP BY 458
Sample Statements 459
Summary 470
Problems for You to Solve 471

CHAPTER 14 Filtering Grouped Data 473

Topics Covered in This Chapter 473
A New Meaning of "Focus Groups" 474
When You Filter Makes a Difference 478
Should You Filter in WHERE or in HAVING? 478
Avoiding the HAVING COUNT Trap 481
Uses for HAVING 486
Sample Statements 487
Summary 496
Problems for You to Solve 496

PART V Modifying Sets of Data 499

CHAPTER 15 Updating Sets of Data 501

Topics Covered in This Chapter 501
What Is an UPDATE? 501
The UPDATE Statement 502
Using a Simple UPDATE Expression 503
A Brief Aside: Transactions 506
Updating Multiple Columns 507
Using a Subquery to Filter Rows 508
Using a Subquery UPDATE Expression 514
Uses for UPDATE 516
Sample Statements 517
Summary 533
Problems for You to Solve 534

CHAPTER 16 Inserting Sets of Data 537

Topics Covered in This Chapter 537
What Is an INSERT? 537
The INSERT Statement 539
Inserting Values 539
Generating the Next Primary Key Value 542
Inserting Data by Using SELECT 544
Uses for INSERT 550
Sample Statements 552
Summary 562
Problems for You to Solve 563

CHAPTER 17 Deleting Sets of Data 567

Topics Covered in This Chapter 567
What Is a DELETE? 567
The DELETE Statement 568
Deleting All Rows 569
Deleting Some Rows 571
Uses for DELETE 575
Sample Statements 576
Summary 583
Problems for You to Solve 584

In Closing 587

APPENDICES 589

A SQL Standard Diagrams 591

B Schema for the Sample Databases 601

C Date and Time Functions 607

D Suggested Reading 615

Index 617

This page intentionally left blank

Foreword

In the 20 years since the database language SQL was adopted as an international standard, and the 25 years since SQL database products appeared on the market, SQL has become the predominant language for storing, modifying, retrieving, and deleting data. Today, a significant portion of the world's data—and the world's economy—is tracked using SQL databases.

SQL is everywhere because it is a very powerful tool for manipulating data. It is in high-performance transaction processing systems. It is behind Web interfaces. I've even found SQL in network monitoring tools and spam firewalls.

Today, SQL can be executed directly, embedded in programming languages, and accessed through call interfaces. It is hidden inside GUI development tools, code generators, and report writers. However visible or hidden, the underlying queries are SQL. Therefore, to understand existing applications and to create new ones, you need to understand SQL.

SQL Queries for Mere Mortals, Second Edition, provides a step-by-step, easy-to-read introduction to writing SQL queries. It includes hundreds of examples with detailed explanations. This book provides the tools you need to understand, modify, and create SQL queries.

As a database consultant and a participant in both the U.S. and international SQL standards committees, I spend a lot of time working with SQL. So, it is with a certain amount of authority that I state, "The authors of this book not only understand SQL, they also understand how to explain it." Both qualities make this book a valuable resource.

Keith W. Hare
Senior Consultant, JCC Consulting, Inc.
Vice Chair, INCITS H2—the USA SQL Standards Committee
Convenor, ISO/IEC JTC1 SC32 WG3—the International
SQL Standards Committee

This page intentionally left blank

Preface

"Language is by its very nature a communal thing;
that is, it expresses never the exact thing but a
compromise—that which is common to you, me, and everybody."
—Thomas Earnest Hulme, *Speculations*

Learning how to retrieve information from or manipulate information in a database is commonly a perplexing exercise. However, it can be a relatively easy task as long as you understand the question you're asking or the change you're trying to make to the database. After you understand the problem, you can translate it into the language used by any database system, which in most cases is Structured Query Language (SQL). You have to translate your request into an SQL statement so that your database system knows what information you want to retrieve or change. SQL provides the means for you and your database system to communicate.

Throughout our many years as database consultants, we've found that the number of people who merely need to retrieve information from a database or perform simple data modifications in a database far outnumber those who are charged with the task of creating programs and applications for a database. Unfortunately, no books focus solely on this subject, particularly from a "mere mortals" viewpoint. There are numerous good books on SQL, to be sure, but most are targeted to database programming and development.

With this in mind, we decided it was time to write a book that would help people learn how to query a database properly and effectively. We produced the first edition of this book in 2000. With this new edition, we also wanted to introduce you to the basic ways to change data in your database using SQL. The result of our decision is in your hands. This book is unique among SQL books in that it focuses on SQL with little regard to any one specific database system implementation. This second edition includes hundreds of new examples, and we included versions of the sample databases using the popular open-source MySQL database system. When you finish reading this book, you'll have the skills you need to retrieve or modify any information you require.

Acknowledgments

Writing a book such as this is always a cooperative effort. There are always editors, colleagues, friends, and relatives willing to lend their support and provide valuable advice when we need it the most. These people continually provide us with encouragement, help us to remain focused, and motivate us to see this project through to the end.

First and foremost, we want to thank our acquisitions editor, Elizabeth Peterson, for prodding us to produce this second edition. Thanks also to Kristin Weinberger for shepherding us along the way. And we can't forget our final acquisitions editor, Chuck Toporek, as well as Romny French and the production staff—they're a great team! Special thanks to Chrysta Meadowbrooke, who did a fabulous job copyediting the final manuscript. She cleaned up lots of inconsistencies and even pointed out some SQL examples that needed fixing! Finally, thanks to editor-in-chief Karen Gettman, who put this team together and kept a watchful eye over the entire process.

Next, we'd like to acknowledge our technical editors, particularly Stephen Forte and Keith Hare. Keith especially spent time working through all the examples, pointing out a few errors, and making suggestions to enhance the text. Thanks once again to all of you for your time and input and for helping us to make this a solid treatise on SQL queries.

Finally, another very special thanks to Keith Hare for providing the Foreword. As the Convenor of the International SQL Standards Committee, Keith is an SQL expert par excellence. We have a lot of respect for Keith's knowledge and expertise on the subject, and we're pleased to have his thoughts and comments at the beginning of our book.

About the Authors

John L. Viescas is an independent consultant with more than 40 years of experience. He began his career as a systems analyst, designing large database applications for IBM mainframe systems. He spent six years at Applied Data Research in Dallas, Texas, where he directed a staff of more than 30 people and was responsible for research, product development, and customer support of database products for IBM mainframe computers. While working at Applied Data Research, John completed a degree in business finance at the University of Texas at Dallas, graduating cum laude.

John joined Tandem Computers, Inc., in 1988, where he was responsible for the development and implementation of database marketing programs in Tandem's U.S. Western Sales region. He developed and delivered technical seminars on Tandem's relational database management system, NonStop SQL, in a geographic area spanning Hawaii to Colorado and Alaska to Arizona. John wrote his first book, *A Quick Reference Guide to SQL* (Microsoft Press, 1989), as a research project to document the similarities in the syntax among the ANSI-86 SQL standard, IBM's DB2, Microsoft's SQL Server, Oracle Corporation's Oracle, and Tandem's NonStop SQL. He wrote the first edition of *Running Microsoft Access* (Microsoft Press, 1992) while on sabbatical from Tandem. He has since written four editions of *Running*, two editions of *Microsoft Office Access Inside Out* (Microsoft Press, 2004 and 2007—the successor to the *Running* series), and *Building Microsoft Access Applications* (Microsoft Press, 2005).

John formed his own company in 1993. He provides information systems management consulting for a variety of small to large businesses around the world, with a specialty in the Microsoft Access and SQL Server database management products. He maintains offices in Nashua, New Hampshire, and Paris, France. He has been recognized as a "Most Valuable Professional" every year since 1993 by Microsoft Product Support Services for his assistance with technical questions on public support forums.

You can visit John's Web site at www.viescas.com or contact him by e-mail at johnv@viescas.com.

Michael J. Hernandez is a veteran database developer with more than 20 years of experience developing applications for a wide variety of clients in diverse industries. Mike specializes in relational database design and is the author of the best-selling database design book *Database Design for Mere Mortals*, Second Edition (Addison-Wesley, 2004). He has worked with SQL throughout his career, developing applications using SQL-based databases such as Microsoft Access and Microsoft SQL Server. He has also been a contributing author and technical editor to various database-related books and periodicals.

Mike became a full-time employee at Microsoft in 2002. He initially was the Community Program Manager for the Visual Studio Tools for Office (VSTO) Team, leading and managing the team's developer community engagement efforts. In 2006, Mike became the Product Manager for VSTO, becoming responsible for helping to guide the strategic future of the product and promoting VSTO to customers and developers via a variety of venues. As he has done so often throughout his career, Mike often speaks at developer events, conferences, and user group meetings across the nation and around the world.

In a previous life, Mike had a career as a musician and performed for audiences far and wide. He attributes both his easygoing presentation style and his ability to connect with an audience to his days as a performer. Ever the musician, Mike formed a band from members of the VSTO team and gets to play his beloved guitar before new crowds and audiences. He still tinkers on his guitar quite a bit, stealing a few minutes here and there between meetings at work. Mike enjoys the little things in life, such as spending long hours at Barnes & Noble, sipping a tall Americano at Starbucks, puffing on a fine cigar, and riding his mountain bike along with his wife, Kendra.

You can contact Mike via e-mail at mjhernandez@msn.com.

Introduction

"I presume you're mortal, and may err."
—James Shirley
The Lady of Pleasure

If you've used a computer more than casually, you have probably used Structured Query Language, or SQL—perhaps without even knowing it. SQL is *the* standard language for communicating with most database systems. Any time you import data into a spreadsheet or perform a merge into a word processing program, you're most likely using SQL in some form or another. Every time you go online to an e-commerce site on the Web and place an order for a book, a recording, a movie, or any of the dozens of other products you can order, there's a very high probability that the code behind the Web page you're using is accessing its databases with SQL. If you need to get information from a database system that uses SQL, you can enhance your understanding of the language by reading this book.

Are You a Mere Mortal?

You might ask, "Who is a *mere mortal*? Me?" The answer is not simple. When we started to write this book, we thought we were experts in the database language called SQL. Along the way, we discovered we were mere mortals too, in several areas. We understood a few specific implementations of SQL very well, but we unraveled many of the complex intricacies of the language as we studied how it is used in many commercial products. So, if you fit any of the following descriptions, you're a mere mortal too!

- If you use computer applications that let you access information from a database system, you're probably a mere mortal. The first time you don't get the information you expected using the query tools built in to

your application, you'll need to explore the underlying SQL statements to find out why.

- If you have recently discovered one of the many available desktop database applications but are struggling with defining and querying the data you need, you're a mere mortal.
- If you're a database programmer who needs to "think out of the box" to solve some complex problems, you're a mere mortal.
- If you're a database guru in one product but are now faced with integrating the data from your existing system into another system that supports SQL, you're a mere mortal.

In short, *anyone* who has to use a database system that supports SQL can use this book. As a beginning database user who has just discovered that the data you need can be fetched using SQL, you will find that this book teaches you all the basics and more. For an expert user who is suddenly faced with solving complex problems or integrating multiple systems that support SQL, this book will provide insights into leveraging the complex abilities of the SQL database language.

About This Book

Everything you read in this book is based on the current International Organization for Standardization (ISO) Standard for the SQL database language (document ISO/IEC 9075-2:2003), as currently implemented in most of the popular commercial database systems. The ISO document was also adopted by the American National Standards Institute (ANSI), so this is truly an international standard. The SQL you'll learn here *is not* specific to any particular software product.

As you'll learn in more detail in Chapter 3, A Concise History of SQL, the SQL Standard defines both more and less than you'll find implemented in most commercial database products. Most database vendors have yet to implement many of the more advanced features, but most do support the core of the standard.

We researched a wide range of popular products to make sure that you can use what we're teaching in this book. When we found parts of the core of the language not supported by some major products, we warned you in the text and showed you alternate ways to state your database requests in standard SQL. When we found significant parts of the SQL Standard supported by only

a few vendors, we introduced you to the syntax and then suggested alternatives.

We have organized this book into five major sections.

- Part I, Relational Databases and SQL, explains how modern database systems are based on a rigorous mathematical model and provides a brief history of the database query language that has evolved into what we know as SQL. We also discuss some simple rules that you can use to make sure your database design is sound.
- Part II, SQL Basics, introduces you to using the SELECT statement, creating expressions, and sorting information with an ORDER BY clause. You'll also learn how to filter data by using a WHERE clause.
- Part III, Working with Multiple Tables, shows you how to formulate queries that draw data from more than one table. Here we show you how to link tables in a query using the INNER JOIN, OUTER JOIN, and UNION operators, and how to work with subqueries.
- Part IV, Summarizing and Grouping Data, discusses how to obtain summary information and group and filter summarized data. Here is where you'll learn about the GROUP BY and HAVING clauses.
- Part V, Modifying Sets of Data, explains how to write queries that modify a set of rows in your tables. In the chapters in this section, you'll learn how to use the UPDATE, INSERT, and DELETE statements.

At the end of the book in the appendices, you'll find syntax diagrams for all the SQL elements you've learned, layouts of the sample databases, a list of date and time manipulation functions implemented in five of the major database systems, and book recommendations to further your study of SQL. There is also a CD containing all the sample databases used throughout the book in several different formats.

What This Book Is Not

Although this book is based on the 2003 SQL Standard that was current at the time of this writing (a 2007/2008 draft standard is in the works), it does not cover every aspect of the standard. In truth, many features in the 2003 SQL Standard won't be implemented for many years—if at all—in the major database system implementations. The fundamental purpose of this book is to

give you a solid grounding in writing queries in SQL. Throughout the book, you'll find us recommending that you "consult your database documentation" for how a specific feature might or might not work. That's not to say we covered only the lowest common denominator for any feature among the major database systems. We do try to caution you when some systems implement a feature differently or not at all.

You'll find it difficult to create other than simple queries using a single table if your database design is flawed. We included a chapter on database design to help you identify when you will have problems, but that one chapter includes only the basic principles. A thorough discussion of database design principles and how to implement a design in a specific database system is beyond the scope of this book.

This book is also not about how to solve a problem in the most efficient way. As you work through many of the later chapters, you'll find we suggest more than one way to solve a particular problem. In some cases where writing a query in a particular way is likely to have performance problems on any system, we try to warn you about it. But each database system has its own strengths and weaknesses. After you learn the basics, you'll be ready to move on to digging into the particular database system you use to learn how to formulate your query solutions so that they run in a more optimal manner.

How to Use This Book

We have designed the chapters in this book to be read in sequence. Each succeeding chapter builds on concepts taught in earlier chapters. However, you can jump into the middle of the book without getting lost. For example, if you are already familiar with the basic clauses in a SELECT statement and want to learn more about JOINs, you can jump right in to Chapters 7 Thinking in Sets, 8 INNER JOINS, and 9 OUTER JOINS.

At the end of many of the chapters you'll find an extensive set of sample problems, their solutions, and sample result sets. We recommend that you study several of the samples to gain a better understanding of the techniques involved and then try solving some of the later samples yourself without looking at the solutions we propose.

Note that where a particular query returns dozens of rows in the result set, we show you only the first few rows in this book to give you an idea of how the answer should look. You might not see the exact same result on your system, however, because each database system that supports SQL has its own

optimizer that figures out the fastest way to solve the query. Also, the first few rows you see returned by your database system might not exactly match the first few we show you unless the query contains an ORDER BY clause that requires the rows to be returned in a specific sequence.

We've also included a complete set of problems for you to solve on your own, which you'll find at the end of most chapters. This gives you the opportunity to really practice what you've just learned in the chapter. Don't worry—the solutions are included in the sample databases on the CD. We've also included hints on those problems that might be a little tricky.

After you have worked your way through the entire book, you'll find the complete SQL diagrams in Appendix A to be an invaluable reference for all the SQL techniques we showed you. You will also be able to use the sample database layouts in Appendix B to help you design your own databases.

Reading the Diagrams Used in This Book

The numerous diagrams throughout the book illustrate the proper syntax for the statements, terms, and phrases you'll use when you work with SQL. Each diagram provides a clear picture of the overall construction of the SQL element currently being discussed. You can also use any of these diagrams as templates to create your own SQL statements or to help you acquire a clearer understanding of a specific example.

All the diagrams are built from a set of core elements and can be divided into two categories: *statements* and *defined terms*. A statement is always a major SQL operation, such as the SELECT statement we discuss in this book, while a defined term is always a component used to build part of a statement, such as a *value expression, a search condition,* or a *conditional expression.* (Don't worry—we'll explain all these terms later in the book.) The only difference between a syntax diagram for a statement and a syntax diagram for a defined term is the manner in which the main syntax line begins and ends. We designed the diagrams with these differences so that you can clearly see whether you're looking at the diagram for an entire statement or a diagram for a term that you might use within a statement. Figure 1 (on page xxviii) shows the beginning and end points for both diagram categories. Aside from this difference, the diagrams are built from the same elements. Figure 2 (on page xxviii) shows an example of each type of syntax diagram and is followed by a brief explanation of each diagram element.

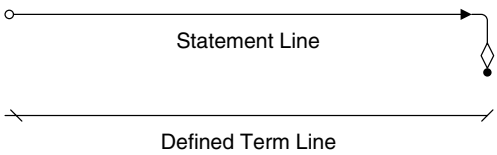


Figure 1 *Syntax line end points for statements and defined terms*

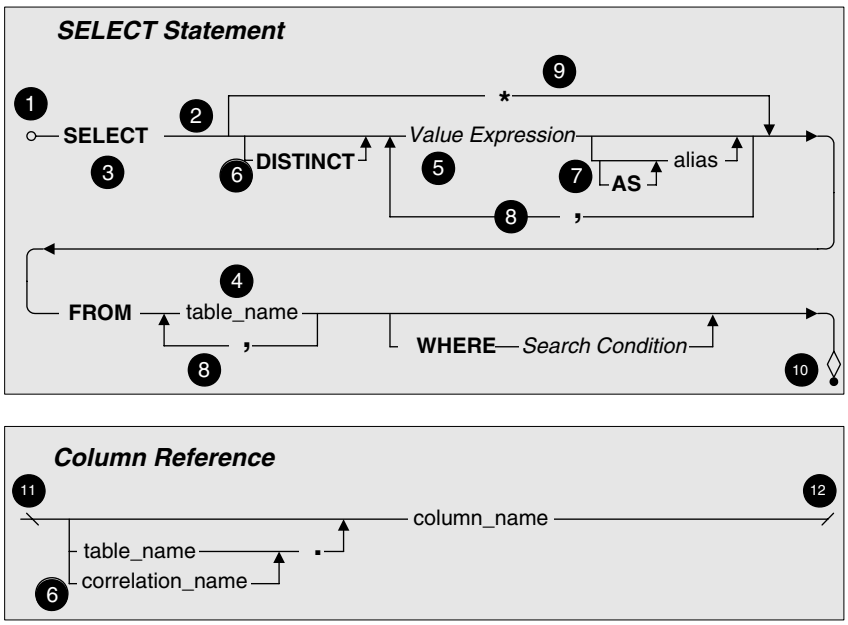


Figure 2 *Sample statement and defined term diagrams*

1. ***Statement start point***—denotes the beginning of the main syntax line for a statement. Any element that appears *directly on* the main syntax line is a *required element*, and any element that appears *below* it is an *optional element*.
2. ***Main syntax line***—determines the order of all required and optional elements for the statement or defined term. Follow this line from left to right (or in the direction of the arrows) to build the syntax for the statement or defined term.
3. ***Keyword(s)***—indicates a major word in SQL grammar that is a required part of the syntax for a statement or defined term. In a diagram, keywords are formatted in capital letters and bold font. (You don't have to worry about typing a keyword in capital letters when you actually write the statement in your database program, but it does make the statement easier to read.)

4. ***Literal entry***—specifies the name of a value you explicitly supply to the statement. A literal entry is represented by a word or phrase that indicates the type of value you need to supply. Literal entries in a diagram are formatted in all lowercase letters.
5. ***Defined term***—denotes a word or phrase that represents some operation that returns a final value to be used in this statement. We'll explain and diagram every defined term you need to know as you work through the book. Defined terms are always formatted in italic letters.
6. ***Optional element***—indicates any element or group of elements that appears below the main syntax line. An optional element can be a statement, keyword, defined term, or literal value and, for purposes of clarity, is placed on its own line. In some cases, you can specify a set of values for a given option, with each value separated by a comma (see number 8). Also, several optional elements have a set of sub-optional elements (see number 7). In general, you read the syntax line for an optional element from left to right, in the same manner that you read the main syntax line. Always follow the directional arrows and you'll be in good shape. Note that some options allow you to specify multiple values or choices, so the arrow will flow from right to left. After you've entered all the items you need, however, the flow will return to normal from left to right. Fortunately, all optional elements work the same way. After we show you how to use an optional element later in the book, you'll know how to use any other optional element you encounter in a syntax diagram.
7. ***Sub-optional element***—denotes any element or group of elements that appears below an optional element. Sub-optional elements allow you to fine-tune your statements so that you can work with more complex problems.
8. ***Option list separator***—indicates that you can specify more than one value for this option and that each value must be separated with a comma.
9. ***Alternate option***—denotes a keyword or defined term that can be used as an alternative to one or more optional elements. The syntax line for an alternate option will bypass the syntax lines of the optional elements it is meant to replace.
10. ***Statement end point***—denotes the end of the main syntax line for a statement.
11. ***Defined term start point***—denotes the beginning of the main syntax line for a defined term.
12. ***Defined term end point***—denotes the end of the main syntax line for a defined term.

Now that you're familiar with these elements, you'll be able to read all the syntax diagrams in the book. And on those occasions when a diagram requires further explanation, we provide you with the information you need to read

the diagram clearly and easily. To help you better understand how the diagrams work, here's a sample SELECT statement that we built using Figure 2.

```
SELECT FirstName, LastName, City, DOB AS DateOfBirth
FROM Students
WHERE City = 'El Paso'
```

This SELECT statement retrieves four columns from the Students table, as we've indicated in the SELECT and FROM clauses. As you follow the main syntax line from left to right, you see that you have to indicate at least one *value expression*. A value expression can be a column name, an expression created using column names, or simply a constant (literal) value that you want to display. You can indicate as many columns as you need with the value expression's *option list separator* (a comma). This is how we were able to use four column names from the Student table. We were concerned that some people viewing the information returned by this SELECT statement might not know what DOB means, so we assigned an *alias* to the DOB column with the value expression's AS sub-option. Finally, we used the WHERE clause to make certain the SELECT statement shows only those students who live in El Paso. (If this doesn't quite make sense to you just now, there's no cause for alarm. You'll learn all this in great detail throughout the remainder of the book.)

You'll find a full set of syntax diagrams in Appendix A. They show the complete and proper syntax for all the statements and defined terms we discuss in the book. If you happen to refer to these diagrams as you work through each chapter, you'll notice a slight disparity between some of the diagrams in a given chapter and the corresponding diagrams in the appendix. The diagrams in the chapters are just simplified versions of the diagrams in the appendix. These simplified versions allow us to explain complex statements and defined terms more easily and give us the ability to focus on particular elements as needed. But don't worry—all the diagrams in the appendix will make perfect sense after you work through the material in the book.

Sample Databases Used in This Book

Bound into the back of the book, you'll find a CD-ROM containing five sample databases that we use for the example queries throughout the book. We've also included diagrams of the database structures in Appendix B: Schema for the Sample Databases.

1. **Sales Orders**. This is a typical order entry database for a store that sells bicycles and accessories. (Every database book needs at least *one* order entry example, right?)
2. **Entertainment Agency**. We structured this database to manage entertainers, agents, customers, and bookings. You would use a similar design to handle event bookings or hotel reservations.
3. **School Scheduling**. You might use this database design to register students at a high school or community college. This database tracks not only class registrations but also which instructors are assigned to each class and what grades the students received.
4. **Bowling League**. This database tracks bowling teams, team members, the matches they played, and the results.
5. **Recipes**. You can use this database to save and manage all your favorite recipes. We even added a few that you might want to try.

On the sample CD, you can find all five sample databases in four different formats.

- Because of the great popularity of the Microsoft Office Access desktop database, we created one set of databases (.mdb file extension) using Microsoft Access 2000 (Version 9.0). We chose Version 9 of this product because it closely supports the current ISO/IEC SQL Standard, and you can open database files in this format using Access 2000, 2002 (XP), 2003, and 2007. You can find these files in the MSAccess subfolder.
- The second format consists of database files (.mdf file extension) created using Microsoft SQL Server 2000. We have also included SQL command files (.sql file extension) and batch files (.bat file extension) that you can use to attach the samples to a Microsoft SQL Server catalog. You can also attach these files to a Microsoft SQL Server 2005 server. You can find these files in the MSSQLServer subfolder. You can obtain a free copy of Microsoft SQL Server 2005 Express Edition at http://msdn.microsoft.com/vstudio/express/sql/download/default.aspx.
- We created the third set of databases using the popular open-source MySQL version 5 database system. You can either point your InnoDB data directory to the MySQL subfolder or use the scripts (.sql file extension) you can also find in that folder to create the database structure, load the data, and create the sample views in your own MySQL data folder. You can obtain a free copy of the community edition of the MySQL database system at http://www.mysql.com/.

- The fourth format is a series of SQL scripts that you can modify and use with any major database system that supports SQL. You can find scripts to define the schema (the tables) of each database, to load the data using INSERT statements, and to create the queries using CREATE VIEW statements in the SQLScripts subfolder. Although we created these scripts using utilities in Microsoft SQL Server, we simplified them to make them generic for use with most database systems.

To install the sample files, see the file ReadMe.txt in the root folder of the sample CD. If you mount the sample CD on an Apple Macintosh system, you will find only the sample files for MySQL and the SQL scripts.

❖ **Note** Although we were very careful to use the most common and simplest syntax for the CREATE TABLE, CREATE INDEX, CREATE CONSTRAINT, and INSERT commands in the sample SQL scripts, you (or your database administrator) might need to modify these files slightly to work with your database system. If you're working with a database system on a remote server, you might need to gain permission from your database administrator to build the samples from the SQL commands we supplied.

For the chapters in Parts II, III, and IV that focus on the SELECT statement, you'll find all the example statements and solutions in the "example" version of each sample database (e.g., SalesOrdersExample, EntertainmentAgency Example). Because the examples in Part V modify the sample data, we created "modify" versions of each of the sample databases (e.g., SalesOrdersModify, EntertainmentAgencyModify). The sample databases for Part V also include additional columns and tables not found in the SELECT examples that enable us to demonstrate certain features of UPDATE, INSERT, and DELETE queries.

"Follow the Yellow Brick Road"

—Munchkin to Dorothy in *The Wizard of Oz*

Now that you've read through the Introduction, you're ready to start learning SQL, right? Well, maybe. At this point, you're still in the house, it's still being tossed about by the tornado, and you haven't left Kansas.

Before you make that jump to Chapter 4, Creating a Sample Query, take our advice and read through the first three chapters. Chapter 1, What Is Relational?, will give you an idea of how the relational database was conceived

and how it has grown to be the most widely used type of database in the industry today. We hope this will give you some amount of insight into the database system you're currently using. In Chapter 2, Ensuring Your Database Structure Is Sound, you'll learn how to fine-tune your data structures so that your data is reliable and, above all, accurate. You're going to have a tough time working with some of the SQL statements if you have poorly designed data structures, so we suggest you read this chapter carefully.

Chapter 3 is literally the beginning of the "yellow brick road." Here you'll learn the origins of SQL and how it evolved into its current form. You'll also learn about some of the people and companies who helped pioneer the language and why there are so many varieties of SQL. Finally, you'll learn how SQL came to be a national and international standard and what the outlook for SQL will be in the years to come.

After you've read these chapters, consider yourself well on your way to Oz. Just follow the road we've laid out through each of the remaining chapters. When you've finished the book, you'll find that you've found the wizard—and he is you.

This page intentionally left blank

Part I

Relational Databases and SQL

This page intentionally left blank

1

What Is Relational?

"Knowledge is the small part of ignorance
that we arrange and classify."
—Ambrose Bierce

Topics Covered in This Chapter

Types of Databases

A Brief History of the Relational Model

Anatomy of a Relational Database

What's in It for You?

Summary

Before delving into the subject of SQL, we need to cover some general background information on the relational database. You'll learn why the relational database was invented, how it is constructed, and why you should use it. This information provides the foundation you need to really understand what SQL is all about and will eventually help to clarify how you can leverage SQL to your best advantage.

Types of Databases

What is a database? As you probably know, a database is an organized collection of data used to model some type of organization or organizational process. It really doesn't matter whether you're using paper or a computer program to collect and store the data. As long as you're collecting and storing data in some organized manner for a specific purpose, you've got a database. Throughout the remainder of this discussion, we'll assume that you're using a computer program to collect and maintain your data.

In general, two types of databases are used in database management: *operational databases* and *analytical databases.*

Operational databases are the backbone of many companies, organizations, and institutions throughout the world today. This type of database is primarily used to collect, modify, and maintain data on a day-to-day basis. The type of data stored is *dynamic*, meaning that it changes constantly and always reflects up-to-the-minute information. Organizations such as retail stores, manufacturing companies, hospitals and clinics, and publishing houses use operational databases because their data is in a constant state of flux.

In contrast, an analytical database stores and tracks historical and time-dependent data. An analytical database is a valuable asset for tracking trends, viewing statistical data over a long period of time, or making tactical or strategic business projections. The type of data stored is *static*, meaning that the data is never (or very rarely) modified, but new data might often be added. The information gleaned from an analytical database reflects a point-in-time snapshot of the data and is usually not up to date. Chemical labs, geological companies, and marketing analysis firms are examples of organizations that use analytical databases.

A Brief History of the Relational Model

Several types of database models exist. Some, such as hierarchical and network, are used only on legacy systems, while others, such as relational, have gained wide acceptance. You might also encounter discussions in other books about object, object-relational, or online analytical processing (OLAP) models. In fact, extensions have been defined in the SQL Standard to support these models, and some commercial database systems have implemented some of the extensions. For our purposes, however, we will focus strictly on the relational model and the core of the international SQL Standard.

In the Beginning . . .

The relational database was first conceived in 1969 and has arguably become the most widely used database model in database management today. The father of the relational model, Dr. Edgar F. Codd (1923–2003), was an IBM research scientist in the late 1960s and was at that time looking into new ways to handle large amounts of data. His dissatisfaction with database models and database products of the time led him to begin thinking of ways to apply the disciplines and structures of mathematics to solve the myriad prob-

lems he had been encountering. A mathematician by profession, he strongly believed that he could apply specific branches of mathematics to solve problems such as data redundancy, weak data integrity, and a database structure's overdependence on its physical implementation.

Dr. Codd formally presented his new relational model in a landmark work titled "A Relational Model of Data for Large Shared Databanks" in June 1970.[1] He based his new model on two branches of mathematics—set theory and first-order predicate logic. Indeed, the name of the model itself is derived from the term *relation*, which is part of set theory. (A widely held misconception is that the relational model derives its name from the fact that tables within a relational database can be related to one another. Now that you know the truth, you'll have a peaceful, restful sleep tonight!) Fortunately, you don't need to know the details of set theory or first-order predicate logic to design and use a relational database. If you use a good database design methodology—such as the one presented in Mike Hernandez's *Database Design for Mere Mortals* (Addison-Wesley, 2004)—you can develop a sound and effective database structure that you can confidently use to collect and maintain any data. (Well, OK, you *do* need to understand a little bit about predicates and set theory to solve more complex problems. We cover the essentials that you need to know about predicates—really a fancy name for a filter—in Chapter 6, Filtering Your Data, and the basics of set theory in Chapter 7, Thinking in Sets.)

Relational Database Software

Since its introduction, the relational model has been the basis for database products known as relational database management systems (RDBMSs). Produced by a variety of vendors, they have gained acceptance over the years by diverse industries and organizations and are used within many types of environments. In the 1970s, mainframe computers used programs such as *System R*, developed by IBM, and *INGRES*, developed at the University of California at Berkeley. The development of RDBMSs for the mainframe continued in the 1980s with programs such as Oracle Corporation's *Oracle* and IBM's *DB2*. The personal computer boom of the mid-1980s gave rise to such programs as Ashton Tate's *dBase*, Ansa Software's *Paradox*, and Microrim's *R:BASE*. When the need to share data among PCs became apparent in the late 1980s and early 1990s, the concept of client/server computing was born along with the idea of centrally located, common data that would be easy to both manage and make secure. This concept gave rise to products such as Oracle's *Oracle 8i* and

[1] *Communications of the ACM,* June 1970, 377–87.

Microsoft's *SQL Server*. Since approximately 1996, there have been more concerted efforts to move database accessibility to the Internet. Software vendors are taking these efforts seriously and are now rising to the occasion by providing products that are more Web-centric, such as Allaire's *Cold Fusion*, Sybase's *Sybase Enterprise Application Studio*, and Microsoft's *Visual Studio*. One of the most popular databases for Web development is the open-source *MySQL* from MySQL AB. Originally designed to run on Linux Web servers, a version of MySQL is also available to run on Microsoft Windows systems.

Anatomy of a Relational Database

According to the relational model, data in a relational database is stored in *relations*, which are perceived by the user as tables. Each relation is composed of *tuples* (records) and *attributes* (fields). A relational database has several other characteristics, which are discussed in this section.

Tables

Tables are the main structures in the database. Each table always represents a single, specific subject. The logical order of records and fields within a table is of absolutely no importance. Every table contains at least one field—known as a *primary key*—that uniquely identifies each of its records. (In Figure 1–1, for example, CustomerID is the primary key of the Customers table.) In fact, data in a relational database can exist independent of the way it is physically stored in the computer because of these last two table characteristics. This is great news for users because they aren't required to know the physical location of a record in order to retrieve its data.

Customers

CustomerID	FirstName	LastName	StreetAddress	City	State	ZipCode
1010	Angel	Kennedy	667 Red River Road	Austin	TX	78710
1011	Alaina	Hallmark	Route 2, Box 203B	Woodinville	WA	98072
1012	Liz	Keyser	13920 S.E. 40th Street	Bellevue	WA	98006
1013	Rachel	Patterson	2114 Longview Lane	San Diego	CA	92199
1014	Sam	Abolrous	611 Alpine Drive	Palm Springs	CA	92263
1015	Darren	Gehring	2601 Seaview Lane	Chico	CA	95926

RECORDS

FIELDS

Figure 1–1 *A sample table*

The subject that a given table represents can be either an *object* or an *event*. When the subject is an object, the table represents something that is tangible, such as a person, place, or thing. Regardless of its type, every object has characteristics that can be stored as data. This data can then be processed in an almost infinite number of ways. Pilots, products, machines, students, buildings, and equipment are all examples of objects that can be represented by a table. Figure 1–1 illustrates one of the most common examples of this type of table.

When the subject of a table is an event, the table represents something that occurs at a given point in time and has characteristics you wish to record. These characteristics can be stored as data and then processed as information in exactly the same manner as a table that represents some specific object. Examples of events you might need to record include judicial hearings, distributions of funds, lab test results, and geological surveys. Figure 1–2 shows an example of a table representing an event that we all have experienced at one time or another—a doctor's appointment.

Patient Visit

PatientID	VisitDate	VisitTime	Physician	BloodPressure	Temperature
92001	2006-05-01	10:30	Ehrlich	120 / 80	98.8
97002	2006-05-01	13:00	Hallmark	112 / 74	97.5
99014	2006-05-02	9:30	Fournier	120 / 80	98.8
96105	2006-05-02	11:00	Hallmark	160 / 90	99.1
96203	2006-05-02	14:00	Hallmark	110 / 75	99.3
98003	2006-05-02	9:30	Fournier	120 / 82	98.6

Figure 1–2 *A table representing an event*

Fields

A field is the smallest structure in the database, and it represents a characteristic of the subject of the table to which it belongs. Fields are the structures actually used to store data. The data in these fields can then be retrieved and presented as information in almost any configuration imaginable. Remember that the quality of the information you get from your data is in direct proportion to the amount of time you've dedicated to ensuring the structural integrity and data integrity of the fields themselves. There is just no way to underestimate the importance of fields.

Every field in a properly designed database contains one and only one value, and its name identifies the type of value it holds. This makes entering data

into a field very intuitive. If you see fields with names such as FirstName, LastName, City, State, and ZipCode, you know exactly what type of value goes into each field. You'll also find it very easy to sort the data by state or to look for everyone whose last name is Viescas.

Records

A record represents a unique instance of the subject of a table. It is composed of the entire set of fields in a table, regardless of whether or not the fields contain any values. Because of the manner in which a table is defined, each record is identified throughout the database by a unique value in the primary key field of that record.

In Figure 1–1, for example, each record represents a unique customer within the table, and the CustomerID field identifies a given customer throughout the database. In turn, each record includes all the fields within the table, and each field describes some aspect of the customer represented by the record. Records are a key factor in understanding table relationships because you need to know how a record in one table relates to other records in another table.

Keys

Keys are special fields that play very specific roles within a table. The type of key determines its purpose within the table. Although a table might contain several types of keys, we will limit our discussion to the two most important ones: the *primary key* and the *foreign key*.

A primary key is a field or group of fields that uniquely identifies each record within a table. (When a primary key is composed of two or more fields, it is known as a *composite primary key*.) The primary key is the most important for two reasons: Its *value* identifies *a specific record* throughout the entire database, and its *field* identifies *a given table* throughout the entire database. Primary keys also enforce table-level integrity and help establish relationships with other tables. Every table in your database should have a primary key.

The AgentID field in Figure 1–3 is a good example of a primary key because it uniquely identifies each agent within the Agents table and helps to guarantee table-level integrity by ensuring nonduplicate records. It is also used to establish relationships between the Agents table and other tables in the database, such as the Entertainers table shown in the example.

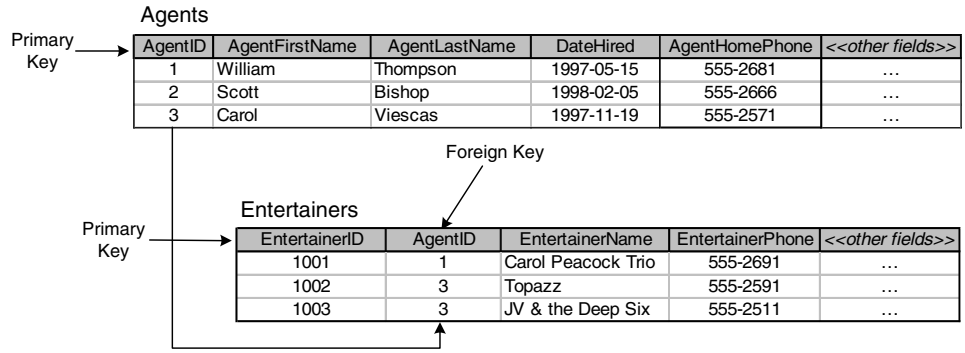


Figure 1–3 *Primary and foreign keys*

When you determine that a pair of tables has a relationship to each other, you typically establish the relationship by taking a copy of the primary key from the first table and inserting it into the second table, where it becomes a foreign key. (The term *foreign key* is derived from the fact that the second table already has a primary key of its own, and the primary key you are introducing from the first table is foreign to the second table.)

Figure 1–3 shows a good example of a foreign key. In this example, AgentID is the primary key of the Agents table, and it is a foreign key in the Entertainers table. As you can see, the Entertainers table already has a primary key—EntertainerID. In this relationship, AgentID is the field that establishes the connection between Agents and Entertainers.

Foreign keys are important not only for the obvious reason that they help establish relationships between pairs of tables but also because they help ensure relationship-level integrity. This means that the records in both tables will always be properly related because the values of a foreign key *must* be drawn from the values of the primary key to which it refers. Foreign keys also help you avoid the dreaded "orphaned records," a classic example of which is an order record without an associated customer. If you don't know who placed the order, you can't process it, and you obviously can't invoice it. That'll throw off your quarterly sales!

Views

A view is a virtual table composed of fields from one or more tables in the database. The tables that comprise the view are known as *base tables*. The relational model refers to a view as virtual because it draws data from base

tables rather than storing any data on its own. In fact, the only information about a view that is stored in the database is its structure.

Views enable you to see the information in your database from many different perspectives, thus providing great flexibility for working with data. You can create views in a variety of ways—they are especially useful when based on multiple related tables. For example, you can create a view that summarizes information such as the total number of hours worked by every carpenter within the downtown Seattle area. Or you can create a view that groups data by specific fields. An example of this type of view is displaying the total number of employees in each city within every state of a specified set of regions. Figure 1–4 presents an example of a typical view.

In many RDBMS programs, a view is commonly implemented and referred to as a *saved query* or, more simply, a *query*. In most cases, a query has all the characteristics of a view, so the only difference is that it is referred to by a different name. (We often wonder if someone in some marketing department had something to do with this.) It's important to note that some vendors are now beginning to call a query by its real name. Regardless of what it's called in your RDBMS program, you'll certainly use views in your database.

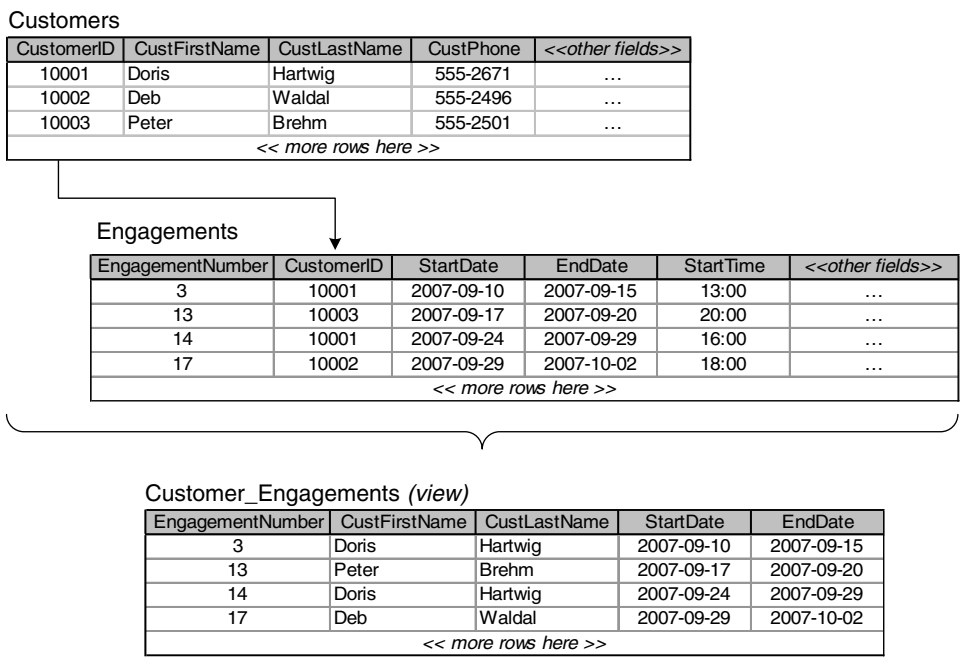


Figure 1–4 *A sample view*

Having said that, the name of this book is *SQL Queries for Mere Mortals*, but we're really focused on teaching you how to build views. As you'll learn in Chapter 2, Ensuring Your Database Structure Is Sound, the correct way to design a relational database is to break up your data so that you have one table per subject or event. But most of the time, you'll want to get information about related subjects or events—which customers placed what orders or what classes are taught by which instructors. To do that, you need to build a view, and you need to know SQL to do that.

Relationships

If records in a given table can be associated in some way with records in another table, the tables are said to have a relationship between them. The manner in which the relationship is established depends on the type of relationship. Three types of relationships can exist between a pair of tables: one-to-one, one-to-many, or many-to-many. Understanding relationships is crucial to understanding how views work and, by definition, how multi-table SQL queries are designed and used. (You'll learn more about this in Part III.)

One-to-One

A pair of tables is related one-to-one when a single record in the first table is related to *only one* record in the second table, and a single record in the second table is related to *only one* record in the first table. In this type of relationship, one table is referred to as the *primary table*, and the other is referred to as the *secondary table*. The relationship is established by taking the primary key of the primary table and inserting it into the secondary table, where it becomes a foreign key. This is a special type of relationship because in many cases the foreign key also acts as the primary key of the secondary table.

An example of a typical one-to-one relationship is shown in Figure 1–5 (on page 12), where Agents is the primary table and Compensation is the secondary table. The relationship between these tables is such that a single record in the Agents table can be related to only one record in the Compensation table, and a single record in the Compensation table can be related to only one record in the Agents table. Note that AgentID is indeed the primary key in both tables but also serves as a foreign key in the secondary table.

The selection of the table that will play the primary role in this type of relationship is purely arbitrary. One-to-one relationships are not very common

Agents

AgentID	AgentFirstName	AgentLastName	DateOfHire	AgentHomePhone	<<other fields>>
1	William	Thompson	1997-05-15	555-2681	…
2	Scott	Bishop	1998-02-05	555-2666	…
3	Carol	Viescas	1997-11-19	555-2571	…

Compensation

Salary	CommissionRate	<<other fields>>
$35,000.00	4.00%	…
$27,000.00	4.00%	…
$30,000.00	5.00%	…

Figure 1–5 *An example of a one-to-one relationship*

and are usually found in cases where a table has been split into two parts for confidentiality purposes.

One-to-Many

When a pair of tables has a one-to-many relationship, a single record in the first table can be related to *many* records in the second table, but a single record in the second table can be related to *only one* record in the first table. This relationship is established by taking the primary key of the table on the "one" side and inserting it into the table on the "many" side, where it becomes a foreign key.

Figure 1–6 shows a typical one-to-many relationship. In this example, a single record in the Entertainers table can be related to *many* records in the Engagements table, but a single record in the Engagements table can be related to *only one* record in the Entertainers table. As you probably have guessed, EntertainerID is a foreign key in the Engagements table.

Entertainers

EntertainerID	EntertainerName	EntertainerPhone	<<other fields>>
1001	Carol Peacock Trio	555-2691	…
1002	Topazz	555-2591	…
1003	JV & the Deep Six	555-2511	…

Engagements

EngagementID	EntertainerID	CustomerID	StartDate	EndDate	<<other fields>>
5	1003	10006	2007-09-11	2007-09-14	…
7	1002	10004	2007-09-11	2007-09-18	…
10	1003	10005	2007-09-17	2007-09-26	…
12	1001	10014	2007-09-18	2007-09-26	…

Figure 1–6 *An example of a one-to-many relationship*

Many-to-Many

A pair of tables is in a many-to-many relationship when a single record in the first table can be related to *many* records in the second table, and a single record in the second table can be related to *many* records in the first table. In order to establish this relationship properly, you must create what is known as a *linking table*. This table provides an easy way to associate records from one table with those of the other and will help to ensure that you have no problems adding, deleting, or modifying any related data. You define a linking table by taking a copy of the primary key of each table in the relationship and using them to form the structure of the new table. These fields actually serve two distinct roles: Together they form the composite primary key of the linking table, and separately they each serve as a foreign key.

A many-to-many relationship that has not been properly established is said to be *unresolved*. Figure 1–7 shows a clear example of an unresolved many-to-many relationship. In this case, a single record in the Customers table can be related to *many* records in the Entertainers table, *and* a single record in the Entertainers table can be related to *many* records in the Customers table.

Customers

CustomerID	CustFirstName	CustLastName	CustPhone	*<<other fields>>*
10001	Doris	Hartwig	555-2671	…
10002	Deb	Waldal	555-2496	…
10003	Peter	Brehm	555-2501	…

Entertainers

EntertainerID	EntertainerName	EntertainerPhone	*<<other fields>>*
1001	Carol Peacock Trio	555-2691	…
1002	Topazz	555-2591	…
1003	JV & the Deep Six	555-2511	…

Figure 1–7 *An unresolved many-to-many relationship*

This relationship is unresolved because of the inherent problem with a many-to-many relationship. The issue is this: How do you easily associate records from the first table with records in the second table? To reframe the question in terms of the tables shown in Figure 1–7, how do you associate a single customer with several entertainers or a specific entertainer with several customers? (If you are running an entertainment booking agency, you certainly hope that any one customer will book multiple entertainers over time and that any one entertainer has more than one customer!) Do you insert a few

customer fields into the Entertainers table? Or do you add several entertainer fields to the Customers table? Either of these approaches is going to create a number of problems when you try to work with related data, not least of which regards data integrity. The solution to this dilemma is to create a linking table in the manner previously stated. By creating and using the linking table, you can properly resolve the many-to-many relationship. Figure 1–8 shows this solution in practice.

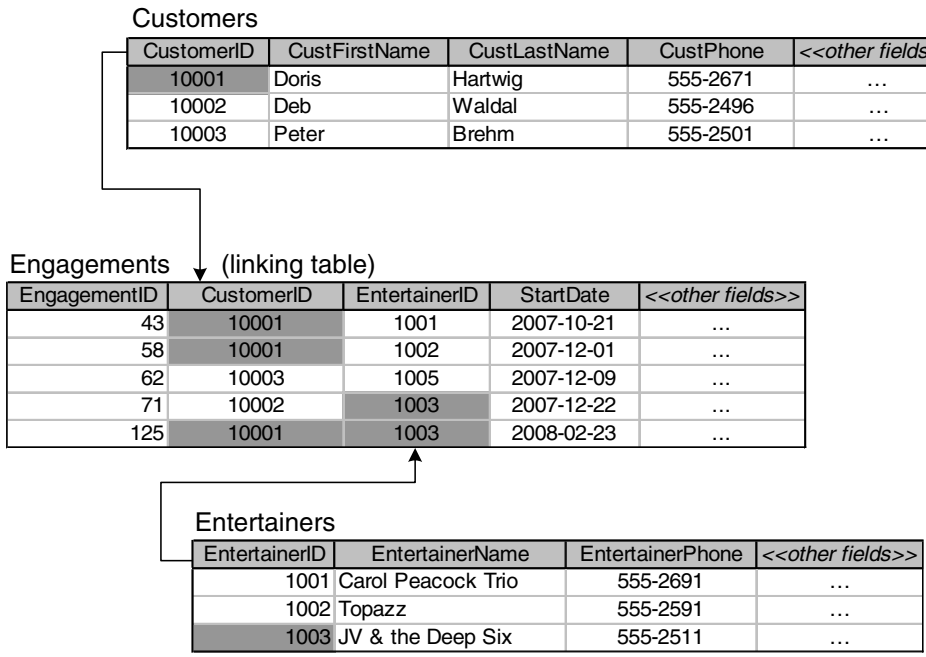


Figure 1–8 *A properly resolved many-to-many relationship*

In Figure 1–8, a linking table was created by taking the CustomerID from the Customers table and the EntertainerID from the Entertainers table and using them as the basis for a new table. As with any other table in the database, the new linking table has its own name—Engagements. In fact, the Engagements table is a good example of a table that stores the information about an event. Entertainer 1003 (JV & the Deep Six) played an engagement for customer 10001 (Doris Hartwig) on February 23. The real advantage of a linking table is that it allows you to associate any number of records from both tables in the relationship. As the example shows, you can now easily associate a given

customer with any number of entertainers or a specific entertainer with any number of customers.

As we stated earlier, understanding relationships will pay great dividends when you begin to work with multi-table SQL queries, so be sure to revisit this section when you begin working on Part III of this book.

What's in It for You?

Why should you be concerned with understanding relational databases? Why should you even care what kind of environment you're using to work with your data? And in addition to all this, what's really in it for you? Here's where the enlightenment starts and the fun begins.

The time you spend learning about relational databases is an investment, and it is to your distinct advantage to do so. You should develop a good working knowledge of the relational database because it's the most widely used data model in existence today. Forget what you read in the trades and what Harry over in the Information Technology Services department told you—a vast majority of the data being used by businesses and organizations is being collected, maintained, and manipulated in relational databases. Yes, there have been extensions to the model, the application programs that work with relational databases have been injected with object orientation, and relational databases have been thoroughly integrated into the Web. But no matter how you slice it, dice it, and spice it, it's still a relational database! The relational database has been around for more than 35 years, it's still going strong, and it's not going be replaced any time in the foreseeable future.

Nearly all commercial database management application software used today is relational. (However, folks such as Dr. Codd, C. J. Date, and Fabian Pascal might seriously question whether any commercial implementation is truly relational!) If you want to be gainfully employed in the database field, you'd better know how to design a relational database and how to implement it using one of the popular RDBMS programs. And now that so many companies and corporations depend on Internet commerce, you'd better have some Web development experience under your belt as well.

Having a good working knowledge of relational databases is helpful in many ways. For instance, the more you know about how relational databases are designed, the easier it will be for you to develop end-user applications for a

given database. You'll also be surprised by how intuitive your RDBMS program will become because you'll understand why it provides the tools it does and how to use those tools to your best advantage. Your working knowledge will be a great asset as you learn how to use SQL because SQL is the standard language for creating, maintaining, and working with a relational database.

Where Do You Go from Here?

Now that you know the importance of learning about relational databases, you must understand that there is a difference between *database theory* and *database design*. Database theory involves the principles and rules that formulate the basis of the relational database model. It is what is learned in the hallowed halls of academia and then quickly dismissed in the dark dens of the real world. But theory is important, nonetheless, because it guarantees that the relational database is structurally sound and that all actions taken on the data in the database have predictable results. On the other hand, database design involves the structured, organized set of processes used to design a relational database. A good database design methodology will help you ensure the integrity, consistency, and accuracy of the data in the database and guarantee that any information you retrieve will be as accurate and up to date as possible.

If you want to design and create enterprise-wide databases, or develop Web-based Internet commerce databases, or begin to delve into data warehousing, you should seriously think about studying database theory. This applies even if you're not going to explore any of these areas but are considering becoming a high-end database consultant. For the rest of you who are going to design and create relational databases on a variety of platforms (which, we believe, is the vast majority of the people reading this book), learning a good, solid database design methodology will serve you well. Always remember that designing a database is relatively easy, but *implementing* a database within a specific RDBMS program on a particular platform is another issue altogether. (Another story, another book, another time.)

There are a number of good database design books on the market. Some, such as Mike Hernandez's companion book *Database Design for Mere Mortals* (Addison-Wesley, 2004), deal only with database design methodologies. Others, such as C. J. Date's *An Introduction to Database Systems* (Addison-Wesley, 2003), mix both theory and design. (Be warned, though, that the books dealing with theory are not necessarily light reading.) After you decide in

which direction you want to go, select and purchase the appropriate books, grab a double espresso (or your beverage of choice), and dig right in. After you become comfortable with relational databases in general, you'll find that you will need to study and become very familiar with SQL.

And that's why you're reading this book.

SUMMARY

We began this chapter with a brief discussion of the different types of databases commonly found today. You learned that organizations working with dynamic data use operational databases, ensuring that the information retrieved is always as accurate and up-to-the-minute as possible. You also learned that organizations working with static data use analytical databases.

We then looked at a brief history of the relational database model. We explained that Dr. E. F. Codd created the model based on specific branches of mathematics and that the model has been in existence for more than 35 years. Database software, as you now know, has been developed for various computer environments and has steadily grown in power, performance, and capability since the 1970s. From the mainframe to the desktop to the Web, RDBMS programs are the backbone of many organizations today.

Next, we looked at an anatomy of a relational database. We introduced you to its basic components and briefly explained their purpose. You learned about the three types of relationships and now understand their importance, not only in terms of the database structure itself but also as they relate to your understanding of SQL.

Finally, we explained why it's to your advantage to learn about relational databases and how to design them. You now know that the relational database is the most common type of database in use today and that just about every database software program you're likely to encounter will be used to support a relational database. You now have some ideas of how to pursue your education on relational database theory and design a little further.

In the next chapter, you'll learn some techniques to fine-tune your existing database structures.

This page intentionally left blank

2

Ensuring Your Database Structure Is Sound

"We shape our buildings: thereafter they shape us."
—Sir Winston Churchill

Topics Covered in This Chapter

Why Is This Chapter Here?

Why Worry about Sound Structures?

Fine-Tuning Fields

Fine-Tuning Tables

Establishing Solid Relationships

Is That All?

Summary

Most of you reading this book are probably working with an existing database structure implemented on your favorite (we hope) RDBMS program. It's hard for us to assume, at this point, whether or not you—or the person who developed the database—really had the necessary knowledge and skills or the time to design the database properly. Assuming the worst, you probably have a number of tables that could use some fine-tuning. Fortunately, you're about to learn some techniques that will help you get your database in shape and will ensure that you can easily retrieve the information you need from your tables.

Why Is This Chapter Here?

You might wonder why we're discussing database design topics in this book and why they're included in a beginning chapter. The reason is simple: If you have a poorly designed database structure, many of the SQL statements you'll

learn to build in the remainder of the book will be, at best, difficult to implement or, at worst, relatively useless. However, if you have a well-designed database structure, the skills you learn in this book will serve you well.

This chapter will not teach you the intricacies of database design, but it will help you get your database in relatively good shape. We highly recommend that you read through this chapter so that you can make certain your table structures are sound.

❖ **Note** It is important to understand that we are about to discuss the *logical* design of the database. We're not teaching you how to create or implement a database in a database management system that supports SQL because, as we mentioned in the Introduction, these subjects are beyond the scope of this book.

Why Worry about Sound Structures?

If your database structure isn't sound, you'll have problems retrieving seemingly simple information from your database, it will be difficult to work with your data, and you'll cringe every time you need to add or delete fields in your tables. Other aspects of the database, such as data integrity, table relationships, and the ability to retrieve accurate information, are affected when you have poorly designed structures. These issues are just the tip of the iceberg. And it goes on! Make sure you have sound structures to avoid all this grief.

You can avoid many of these problems if you properly design your database from the beginning. Even if you've already designed your database, all is not lost. You can still apply the following techniques and gain the benefits of a sound structure. However, you must be aware that the quality of your final structures is in direct proportion to the amount of time you invest in fine-tuning them. The more care and patience you give to applying the techniques, the more you can guarantee your success.

Let's now turn to the first order of business in shaping up your structures: working with the fields.

Fine-Tuning Fields

Because fields are the most basic structures in a database, you must ensure that they are in tip-top shape before you begin fine-tuning the tables as a whole. In many cases, fixing the fields will eliminate a number of existing problems with a given table and help you avoid any potential problems that might have arisen.

What's in a Name? (Part One)

As you learned in the previous chapter, a field represents a characteristic of the subject of the table to which it belongs. If you give the field an appropriate name, you should be able to identify the characteristic it's supposed to represent. A name that is ambiguous, vague, or unclear is a sure sign of trouble and suggests that the purpose of the field has not been carefully thought out. Use the following checklist to test each of your field names.

- *Is the name descriptive and meaningful to your entire organization?* If users in several departments are going to work with this database, make certain you choose a name that is meaningful to everyone who accesses this field. Semantics is a funny thing, and if you use a word that has a different meaning to different groups of people, you're just inviting trouble.
- *Is the field name clear and unambiguous?* PhoneNumber is a field name that can be very misleading. What kind of phone number is this field supposed to represent? A home phone? A work phone? A cellular phone? Learn to be specific. If you need to record each of these types of phone numbers, then create HomePhone, WorkPhone, and CellPhone fields.

  In addition to making your field names clear and unambiguous, be sure that you don't use the same field name in several tables. Let's say you have three tables called Customers, Vendors, and Employees. No doubt you will have City and State fields in each of these tables, and the fields will have the same names in all three tables. There isn't a problem with this until you have to refer to one particular field. How do you distinguish between, say, the City field in the Vendors table, the City field in the Customers table, and the City field in the Employees table? The answer is simple: Add a short prefix to each of the field names. For example, use the name VendCity in the Vendors table, CustCity in the

Customers table, and EmpCity in the Employees table. Now you can easily make a clear reference to any of these fields. (You can use this technique on any generic field such as FirstName, LastName, and Address.)

Here's the main thing to remember: Make sure that each field in your database has a unique name and that it appears only once in the entire database structure. The only exception to this rule is when a field is being used to establish a relationship between two tables.

- *Did you use an acronym or abbreviation as a field name?* If you did, change it! Acronyms can be hard to decipher and are easily misunderstood. Imagine seeing a field named CAD_SW. How would you know what the field represents? Use abbreviations sparingly, and handle them with care. Use an abbreviation only if it supplements or enhances the field name in a positive manner. It shouldn't detract from the meaning of the field name.
- *Did you use a name that implicitly or explicitly identifies more than one characteristic?* These types of names are easy to spot because they typically use the words *and* or *or*. Field names that contain a back slash (\), a hyphen (-), or an ampersand (&) are dead giveaways as well. If you have fields with names such as Phone\Fax or Area or Location, review the data that they store and determine whether you need to deconstruct them into smaller, distinct fields.

❖ **Note** The SQL Standard defines a *regular identifier* as a name that must begin with a letter and can contain only letters, numbers, and the underscore character. Spaces *are not* allowed. It also defines a *delimited identifier* as a name—surrounded with double quotes—that must start with a letter and can contain letters, numbers, the underscore character, spaces, and a very specific set of special characters. Because many SQL implementations support only the regular identifier naming convention, we recommend that you use this naming convention exclusively for your field names.

After using this checklist to revise your field names, you have one task left: Make certain you use the singular form of the field name. A field with a plural name such as Categories implies that it might contain two or more values for any given record, which is not a good idea. A field name is singular because it represents a single characteristic of the subject of the table to which it belongs. A table name, on the other hand, is plural because it represents a collection of similar objects or events. You can distinguish table names from field names quite easily when you use this naming convention.

Smoothing Out the Rough Edges

Now that you've straightened out the field names, let's focus on the structure of the field itself. Although you might be fairly sure that your fields are sound, you can do a few things to make certain they're built as efficiently as possible. Test your fields against the following checklist to determine whether or not your fields need a little more work.

- *Make sure the field represents a specific characteristic of the subject of the table.* The idea here is to determine whether the field truly belongs in the table. If it isn't germane to the table, remove it, or perhaps move it to another table. The only exceptions to this rule occur when the field is being used to establish a relationship between this table and other tables in the database or when it has been added to the table in support of some task required by a database application. For example, in the Classes table in Figure 2–1, the StaffLastName and StaffFirstName fields are unnecessary because of the presence of the

Staff

StaffID	StaffFirstName	StaffLastName	StaffStreetAddress	StaffCity	StaffState	*<<other fields>>*
98014	Peter	Brehm	722 Moss Bay Blvd.	Kirkland	WA	...
98019	Mariya	Sergienko	901 Pine Avenue	Portland	OR	...
98020	Jim	Glynn	13920 S.E. 40th Street	Bellevue	WA	...
98021	Tim	Smith	30301 166th Ave. N.E.	Seattle	WA	...
98022	Carol	Viescas	722 Moss Bay Blvd.	Kirkland	WA	...
98023	Alaina	Hallmark	Route 2, Box 203 B	Woodinville	WA	...

Classes

ClassID	Class	ClassroomID	StaffID	StaffLastName	StaffFirstName	*<<other fields>>*
1031	Art History	1231	98014	Brehm	Peter	...
1030	Art History	1231	98014	Brehm	Peter	...
2213	Biological Principles	1532	98021	Smith	Tim	...
2005	Chemistry	1515	98019	Sergienko	Mariya	...
2001	Chemistry	1519	98023	Hallmark	Alaina	...
1006	Drawing	1627	98020	Glynn	Jim	...
2907	Elementary Algebra	3445	98022	Viescas	Carol	...

Figure 2–1 *A table with unnecessary fields*

StaffID field. StaffID is being used to establish a relationship between the Classes table and the Staff table, and you can view data from both tables simultaneously by using a view or an SQL SELECT query. If you have unnecessary fields in your tables, you can either remove them completely or use them as the basis of a new table if they don't appear anywhere else in the database structure. (We'll show you how to do this later in this chapter.)

- *Make certain that the field contains only a single value.* A field that can potentially store several instances of the same type of value is known as a *multivalued* field. (For example, a field that contains multiple phone numbers is a multivalued field.) Likewise, a field that can potentially store two or more *distinct* values is known as a *multipart* field. (For example, a field that contains both an item number and an item description is a multipart field.) Multivalued and multipart fields can wreak havoc in your database, especially when you try to edit, delete, or sort the data. When you ensure that each field stores only a single value, you go a long way toward guaranteeing data integrity and accurate information. But for the time being, just try to identify any multivalued or multipart fields and make note of them. You'll learn how to resolve them in the next section.
- *Make sure the field does not store the result of a calculation or concatenation.* Calculated fields are not allowed in a properly designed table. The issue here is the value of the calculated field itself. A field, unlike a cell in a spreadsheet, does not store an actual calculation. When the value of any part of the calculation changes, the result value stored in the field is not updated. The only ways to update the value are to do so manually or to write some procedural code that will do it automatically. Either way, it is incumbent on the user or you, the developer, to make certain the value is updated. The preferred way to work with a calculation, however, is to incorporate it into a SELECT statement. You'll learn the advantages of dealing with calculations in this manner when you get to Chapter 5, Getting More Than Simple Columns.
- *Make certain the field appears only once in the entire database.* If you've made the common mistake of inserting the same field (for example, CompanyName) into several tables within the database, you're going to have a problem with inconsistent data. This occurs when you change the value of this field in one table and forget to make the same modification wherever else the field appears. Avoid this problem

entirely by ensuring that a field appears only once in the entire database structure. (The only exception to this rule is when you're using a field to establish a relationship between two tables.)

❖ **Note** The most recent versions of some commercially available database management systems allow you to define a column that is the result of a calculated expression. If your database system has this feature, you can define calculated fields, but be aware that the database system requires additional resources to keep the calculated value current any time the value of one of the fields in the expression changes.

Resolving Multipart Fields

As we mentioned earlier, multipart and multivalued fields will wreak havoc with data integrity, so you need to resolve them in order to avoid any potential problems. Deciding which to resolve first is purely arbitrary, so we'll begin with multipart fields.

You'll know if you have a multipart field by answering some very simple questions: "Can I take the current value of this field and break it up into smaller, more distinct parts?" "Will I have problems extracting a specific piece of information because it is buried in a field containing other information?" If your answer to either question is "Yes," you have a multipart field. Figure 2–2 shows a poorly designed table with several multipart fields.

Customers

CustomerID	CustomerName	StreetAddress	PhoneNumber	*<<other fields>>*
1001	Suzanne Viescas	15127 NE 24th, #383, Redmond, WA 98052	425 555-2686	...
1002	William Thompson	122 Spring River Drive, Duvall, WA 98019	425 555-2681	...
1003	Gary Hallmark	Route 2, Box 203B, Auburn, WA 98002	253 555-2676	...
1004	Robert Brown	672 Lamont Ave, Houston, TX 77201	713 555-2491	...
1005	Dean McCrae	4110 Old Redmond Rd., Redmond, WA 98052	425 555-2506	...
1006	John Viescas	15127 NE 24rh, #383, Redmond, WA 98052	425 555-2511	...
1007	Mariya Sergienko	901 Pine Avenue, Portland, OR 97208	503 555-2526	...
1008	Neil Patterson	233 West Valley Hwy, San Diego, CA 92199	619 555-2541	...

MULTIPART FIELDS

Figure 2–2 *A table with multipart fields*

The Customers table contains two multipart fields: CustomerName, and Street Address. There's also one field that is potentially multipart: PhoneNumber. How can you sort by last name or ZIP Code? You can't because these values are embedded in fields that contain other information. You can see that each field can be broken into smaller fields. For example, CustomerName can be broken into two distinct fields—CustFirstName and CustLastName. (Note that we're using the naming convention discussed earlier in this chapter when we add the prefix Cust to the FirstName and LastName fields.) When you identify a multipart field in a table, determine how many parts there are to the value it stores, and then break the field into as many smaller fields as appropriate. Figure 2–3 shows how to resolve two of the multipart fields in the Customers table.

Customers

CustomerID	CustFirstName	CustLastName	CustAddress	CustCity	CustState	CustZipcode
1001	Suzanne	Viescas	15127 NE 24th, #383	Redmond	WA	98052
1002	William	Thompson	122 Spring River Drive	Duvall	WA	98019
1003	Gary	Hallmark	Route 2, Box 203B	Auburn	WA	98002
1004	Robert	Brown	672 Lamont Ave	Houston	TX	77201
1005	Dean	McCrae	4110 Old Redmond Rd.	Redmond	WA	98052
1006	John	Viescas	15127 NE 24th, #383	Redmond	WA	98052
1007	Mariya	Sergienko	901 Pine Avenue	Portland	OR	97208
1008	Neil	Patterson	233 West Valley Hwy	San Diego	CA	92199

Figure 2–3 *The resolution of the multipart fields in the Customers table*

❖ **Note** Along with breaking down CustomerName and StreetAddress, it might also be a good idea in a database storing phone numbers in North America to break PhoneNumber into two distinct fields—area code and the local phone number. In other countries, separating out the city code portion of the phone number might be useful. In truth, most business databases store a phone number as one field, but separating out the area or city code might be important for databases that analyze demographic data. Unfortunately, we couldn't demonstrate this in Figure 2–3 due to space limitations.

Sometimes you might have difficulty recognizing a multipart field. Take a look at the Instruments table shown in Figure 2–4. At first glance, there do not seem to be any multipart fields. On closer inspection, however, you will see that InstrumentID is actually a multipart field. The value stored in this field represents two distinct pieces of information: the category to which the instrument belongs—such as AMP (amplifier), GUIT (guitar), and MFX (multi-effects

unit)—and its identification number. You should separate these two values and store them in their own fields to ensure data integrity. Imagine the difficulty of updating this field if the MFX category changed to MFU. You would have to write code to parse the value in this field, test for the existence of MFX, and then replace it with MFU if it does exist within the parsed value. It's not so much that you *couldn't* do this, but you'd definitely be working harder than necessary, and you shouldn't have to go through this at all if your database is properly designed. When you have fields such as the one in this example, break them into smaller fields so that you will have sound, efficient field structures.

Instruments

InstrumentID	Manufacturer	InstrumentDescription	*<<other fields>>*
GUIT2201	Fender	Fender Stratocaster	...
MFX3349	Zoom	Player 2100 Multi-Effects	...
AMP1001	Marshall	JCM 2000 Tube Super Lead	...
AMP5590	Crate	VC60 Pro Tube Amp	...
SFX2227	Dunlop	Cry Baby Wah-Wah	...
AMP2766	Fender	Twin Reverb Reissue	...

Figure 2–4 *An example of a subtle multipart field*

Resolving Multivalued Fields

Resolving multipart fields is not very hard at all, but resolving multivalued fields can be a little more difficult and will take some work. Fortunately, identifying a multivalued field is easy. Almost without exception, the data stored in this type of field contains a number of commas, semicolons, or other common separator characters. The separator characters are used to separate the various values within the field itself. Figure 2–5 shows an example of a multivalued field.

Pilots

PilotID	PilotFirstName	PilotLastName	HireDate	Certifications	*<<other fields>>*
25100	Sam	Alborous	1994-07-11	727, 737, 757, MD80	...
25101	Jim	Wilson	1994-05-01	737, 747, 757	...
25102	David	Smith	1994-09-11	757, MD80, DC9	...
25103	Kathryn	Patterson	1994-07-11	727, 737, 747, 757	...
25104	Michael	Hernandez	1994-05-01	737, 757, DC10	...
25105	Kendra	Bonnicksen	1994-09-11	757, MD80, DC9	...

Figure 2–5 *A table with a multivalued field*

In this example, each pilot is certified to fly any number of planes, and those certifications are stored in a single field called Certifications. The manner in which the data is stored in this field is very troublesome because you are bound to encounter the same type of data integrity problems associated with multipart fields. When you look at the data more closely, you'll see that it will be difficult for you to perform searches and sorts on this field in an SQL query. Before you can resolve this field in the appropriate manner, you must first understand the true relationship between a multivalued field and the table to which it is originally assigned.

The values in a multivalued field have a many-to-many relationship with every record in its parent table: One specific value in a multivalued field can be associated with any number of records in the parent table, and a single record in the parent table can be associated with any number of values in the multivalued field. In Figure 2–5, for example, a specific aircraft in the Certifications field can be associated with any number of pilots, and a single pilot can be associated with any number of aircraft in the Certifications field. You resolve this many-to-many relationship as you would any other many-to-many relationship within the database—with a linking table.

To create the linking table, use the multivalued field and a *copy* of the primary key field from the original table as the basis for the new table. Give the new linking table an appropriate name, and designate both fields as a composite primary key. (In this case, the combination of the values of both fields will uniquely identify each record within the new table.) Now you can associate the values of both fields in the linking table on a one-to-one basis. Figure 2–6 shows an example of this process using the Pilots table shown in Figure 2–5.

Contrast the entries for Sam Alborous (PilotID 25100) in both the old Pilots table and the new Pilot_Certifications table. The major advantage of the new linking table is that you can now associate *any* number of certifications with a single pilot. Asking certain types of questions is now much easier as well. For example, you can determine which pilots are certified to fly a Boeing 747 aircraft or retrieve a list of certifications for a specific pilot. You'll also find that you can sort the data in any order you wish, without any adverse effects.

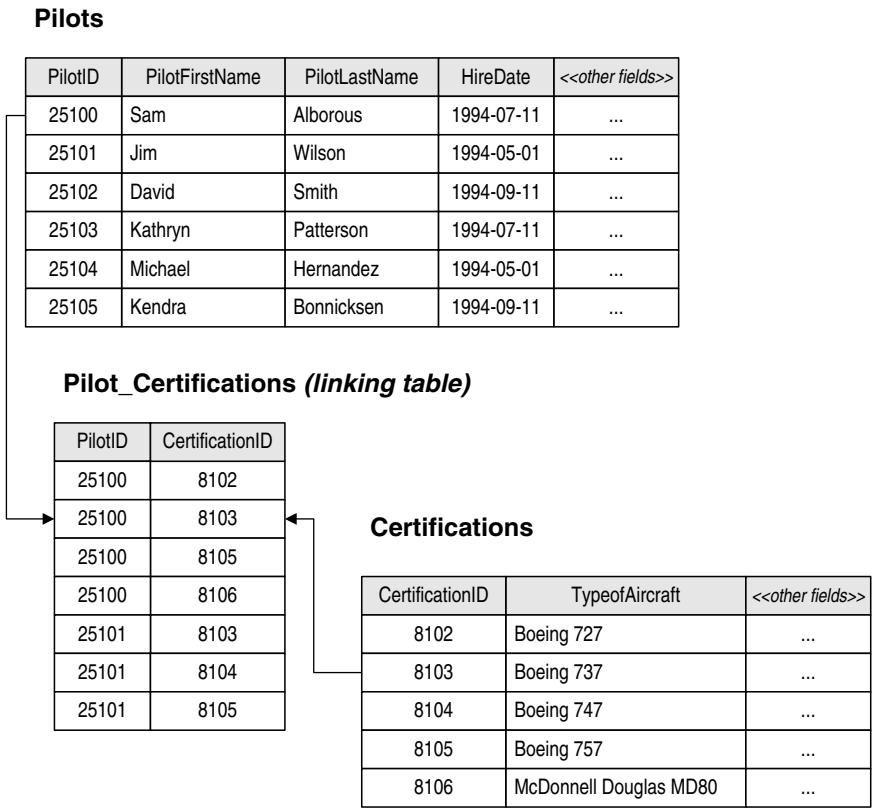


Figure 2–6 *Resolving a multivalued field by using a linking table*

❖ **Note** Some database management systems—most notably Microsoft Office Access 2007— allow you to explicitly define multivalued fields. The database system does this, however, by creating a hidden system table similar to the linking table shown in Figure 2–6. Frankly, we like to see and control our table designs, so we recommend that you create the correct data structures yourself rather than depend on a feature in your database system.

When you follow the procedures presented in this section, your fields will be in good shape. Now that you've refined the fields, let's turn to our second order of business and take a look at the table structures.

Fine-Tuning Tables

Tables serve as the basis for any SQL query you create. You'll soon find that poorly designed tables pose data integrity problems and are difficult to work with when you create multi-table SQL queries. Because of this, you must make certain that your tables are structured as efficiently as possible so that you can easily retrieve the information you need.

What's in a Name? (Part Two)

In the section on fields, you learned how important it is for a field to have an appropriate name and why you should give serious thought to naming your fields. In this section, you'll learn that the same applies to tables as well. By definition, a table should represent a single subject. If it represents more than one subject, it should be divided into smaller tables. The name of the table must clearly identify the subject the table represents. You can be confident that the subject of the table has not been carefully thought out if a table name is ambiguous, vague, or unclear. Make sure your table names are sound by checking them against the following checklist.

- *Is the name unique and descriptive enough to be meaningful to your entire organization?* Giving your table a unique name ensures that each table in the database represents a different subject and that everyone in the organization will understand what the table represents. Defining a unique and descriptive name does take some work on your part, but it's well worth the effort in the long run.
- *Does the name accurately, clearly, and unambiguously identify the subject of the table?* When the table name is vague or ambiguous, you can bet that the table represents more than one subject. For example, Dates is a vague table name. It's hard to determine exactly what this table represents unless you have a description of the table at hand. For example, let's say this table appears in a database used by an entertainment agency. If you inspect this table closely, you'll probably find that it contains dates for client meetings and booking dates for the agency's stable of entertainers. This table clearly represents two subjects. In this case, divide the table into two new tables and give each table an appropriate name, such as Client_Meetings and Entertainer_Schedules.

- *Does the name contain words that convey physical characteristics?* Avoid using words such as *File*, *Record*, and *Table* in the table name because they introduce a level of confusion that you don't need. A table name that includes this type of word is very likely to represent more than one subject. Consider the name Employee_Record. On the surface, there doesn't appear to be any problem with this name. But when you think about what an employee record is supposed to represent, you'll realize that there are potential problems. The name contains a word that we're trying hard to avoid, and it potentially represents three subjects: employees, departments, and payroll. With this in mind, split the original table (Employee_Record) into three new tables, one for each of the three subjects.
- *Did you use an acronym or abbreviation as a table name?* If the answer to this question is "Yes," change the name right now! Abbreviations rarely convey the subject of the table, and acronyms are usually hard to decipher. For example, say your company database has a table named SC. How do you know what the table represents without knowing the meaning of the letters themselves? The fact is that you can't easily identify the subject of the table. What's more, you might find that the table means different things to different departments in the company. (Now, this is scary.) The folks in Personnel think it stands for Steering_Committees; the Information Systems staff believes it to be System_Configurations; and the people in Security insist that it represents Security_Codes. This example clearly illustrates why you should avoid using abbreviations and acronyms in a table name.
- *Did you use a name that implicitly or explicitly identifies more than one subject?* This is one of the most common mistakes you can make with a table name, and it is relatively easy to identify. This type of name typically contains the words *and* or *or* and characters such as the back slash (\), hyphen (-), or ampersand (&). Facility\Building and Department or Branch are typical examples. When you name a table in this manner, you must clearly identify whether it truly represents more than one subject. If it does, deconstruct it into smaller tables, and then give the new tables appropriate names.

❖ **Note** Remember that the SQL Standard defines a *regular identifier* as a name that must begin with a letter and can contain only letters, numbers, and the underscore character. Spaces *are not* allowed. It also defines a *delimited identifier* as a name—surrounded with double quotes—that must start with a letter and can contain letters, numbers, the underscore character spaces, and a very specific set of special characters. Because many SQL implementations support only the regular identifier naming convention, we recommend that you use this naming convention exclusively for your table names.

After you've finished revising your table names, you have one more task to perform: Check each table name again once more, and make certain you used the plural form of the name. You use the plural form because a table stores a *collection of instances* of the subject of the table. For example, an Employees table stores the data not for only one employee but for many employees. Using the plural form also helps you to distinguish a table name from a field name.

Ensuring a Sound Structure

Let's focus on the table structures now that you've revised the table names. It's imperative that the tables are properly designed so that you can efficiently store data and retrieve accurate information. The time you spend ensuring your tables are well built will pay dividends when you need to create complex multi-table SQL queries. Use the following checklist to determine whether your table structures are sound.

- *Make sure the table represents a single subject.* Yes, we know, we've said this a number of times already, but we can't overemphasize this point. As long as you guarantee that each of your tables represents a single subject, you greatly reduce the risk of potential data integrity problems. Also remember that the subject represented by the table can be an object or event. By "object" we mean something that is tangible, such as employees, vendors, machines, buildings, or departments. On the other hand, an "event" is something that happens at a given point in time that has characteristics you want to record. The best example of an event that everyone can relate to is a doctor's appointment. Although you can't explicitly touch a doctor's appointment, it does have characteristics that you need to record, such as the appointment

date, the appointment time, the patient's blood pressure, and the patient's temperature.

- *Make certain each table has a primary key.* You must assign a primary key to each table for two reasons. First, the primary key uniquely identifies each record within a table, and second, it is used in establishing table relationships. If you do not assign a primary key to each table, you will eventually have data integrity problems and problems with some types of multi-table SQL queries. You'll learn some tips on how to define a proper primary key later in this chapter.
- *Make sure the table does not contain any multipart or multivalued fields.* Theoretically, you should have resolved these issues when you refined the field structures. Nonetheless, it's still a good idea to review the fields one last time to ensure that you've completely removed each and every multipart or multivalued field.
- *Make sure there are no calculated fields in the table.* Although you might believe that your current table structures are free of calculated fields, you might have overlooked one or two during the field refinement process. This is a good time to take another look at the table structures and remove any calculated fields you might have missed.
- *Make certain the table is free of any unnecessary duplicate fields.* One of the hallmarks of a poorly designed table is the inclusion of duplicate fields from other tables. You might feel compelled to add duplicate fields to a table for one of two reasons: to provide reference information or to indicate multiple occurrences of a particular type of value. These duplicate fields raise various difficulties when you work with the data and attempt to retrieve information from the table. Let's now take a look at how to deal with duplicate fields.

Resolving Unnecessary Duplicate Fields

Possibly the hardest part of ensuring well-built structures is dealing with duplicate fields. Here are a couple of examples that demonstrate how to properly resolve tables that contain duplicate fields.

Figure 2–7 (on page 34) illustrates an example of a table containing duplicate fields that supply reference information.

Staff

StaffID	StaffFirstName	StaffLastName	StaffStreetAddress	StaffCity	StaffState	*<<other fields>>*
98014	Peter	Brehm	722 Moss Bay Blvd.	Kirkland	WA	...
98019	Mariya	Sergienko	901 Pine Avenue	Portland	OR	...
98020	Jim	Glynn	13920 S.E. 40th Street	Bellevue	WA	...
98021	Tim	Smith	30301 166th Ave. N.E.	Seattle	WA	...
98022	Carol	Viescas	722 Moss Bay Blvd.	Kirkland	WA	...
98023	Alaina	Hallmark	Route 2, Box 203 B	Woodinville	WA	...

These fields are unnecessary

Classes

ClassID	Class	ClassroomID	StaffID	StaffLastName	StaffFirstName	*<<other fields>>*
1031	Art History	1231	98014	Brehm	Peter	...
1030	Art History	1231	98014	Brehm	Peter	...
2213	Biological Principles	1532	98021	Smith	Tim	...
2005	Chemistry	1515	98019	Sergienko	Mariya	...
2001	Chemistry	1519	98023	Hallmark	Alaina	...
1006	Drawing	1627	98020	Glynn	Jim	...
2907	Elementary Algebra	3445	98022	Viescas	Carol	...

Figure 2–7 *A table with duplicate fields added for reference information*

In this case, StaffLastName and StaffFirstName appear in the Classes table so that a person viewing the table can see the name of the instructor for a given class. However, these fields are unnecessary because of the one-to-many relationship that exists between the Classes and Staff tables. (A single staff member can teach any number of classes, but a single class is taught by a specific staff member.) StaffID establishes the relationship between these tables, and the relationship itself lets you view data from both tables simultaneously in an SQL query. With this in mind, you can confidently remove the StaffLastName and StaffFirstName fields from the Classes table without any adverse effects. Figure 2–8 shows the revised Classes table structure.

Staff

StaffID	StaffFirstName	StaffLastName	StaffStreetAddress	StaffCity	StaffState	*<<other fields>>*
98014	Peter	Brehm	722 Moss Bay Blvd.	Kirkland	WA	...
98019	Mariya	Sergienko	901 Pine Avenue	Portland	OR	...
98020	Jim	Glynn	13920 S.E. 40th Street	Bellevue	WA	...
98021	Tim	Smith	30301- 166th Ave. N.E.	Seattle	WA	...
98022	Carol	Viescas	722 Moss Bay Blvd.	Kirkland	WA	...
98023	Alaina	Hallmark	Route 2, Box 203 B	Woodinville	WA	...

Classes

ClassID	Class	ClassroomID	StaffID	*<<other fields>>*
1031	Art History	1231	98014	...
1030	Art History	1231	98014	...
2213	Biological Principles	1532	98021	...
2005	Chemistry	1515	98019	...
2001	Chemistry	1519	98023	...
1006	Drawing	1627	98020	...
2907	Elementary Algebra	3445	98022	...

Figure 2–8 *Resolving the duplicate reference fields*

Keeping these unnecessary fields in the table automatically introduces a major problem with inconsistent data. You must ensure that the values of the StaffLastName and StaffFirstName fields in the Classes table always match their counterparts in the Staff table. For example, say a female staff member marries and decides to use her married name as her legal name from that day forward. Not only do you have to be certain to make the appropriate change to her record in the Staff table, but you must ensure that every occurrence of her name in the Classes table changes as well. Again, it's possible to do this (at least, technically), but you're working much harder than is necessary. Besides, one of the major premises behind using a relational database is that you should enter a piece of data only once in the entire database. (The only exception to this rule is when you're using a field to establish a relationship between two tables.) As always, the best course of action is to remove all duplicate fields from the tables in your database.

Figure 2–9 shows another clear example of a table containing duplicate fields. This example illustrates how duplicate fields are mistakenly used to indicate multiple occurrences of a particular type of value. In this case, the three Committee fields are ostensibly used to record the names of the committees in which the employee participates.

Employees

EmployeeID	EmpLastName	EmpFirstName	Committee1	Committee2	Committee3	*<<other fields>>*
7004	Gehring	Darren	Steering			...
7005	Kennedy	John	ISO 9000	Safety		...
7006	Thompson	Sarah	Safety	ISO 9000	Steering	...
7007	Wilson	Jim				...
7008	Seidel	Manuela	ISO 9000			...
7009	Smith	David	Steering	Safety	ISO 9000	...
7010	Patterson	Neil				...
7011	Viescas	Michael	ISO 9000	Steering	Safety	...

Figure 2–9 *A table with duplicate fields used to indicate multiple occurrences of a particular type of value*

It's relatively easy to see why these duplicate fields will create problems. One problem concerns the actual number of Committee fields in the table. What if a few employees end up belonging to four committees? For that matter, how can you tell exactly how many Committee fields you're going to need? If it turns out that several employees participate in more than three committees, you'll need to add more Committee fields to the table.

A second problem pertains to retrieving information from the table. How do you retrieve those employees who are currently in the ISO 9000 committee? It's not impossible, but you'll have difficulty retrieving this information. You must execute three separate queries (or build a search condition that tests three separate fields) in order to answer the question accurately because you cannot be certain in which of the three Committee fields the value ISO 9000 is stored. Now you're expending more time and effort than is truly necessary.

A third problem concerns sorting the data. You cannot sort the data by committee in any practical fashion, and there's no way that you'll get the committee names to line up correctly in alphabetical order. Although these might seem like minor problems, they can be quite frustrating when you're trying to get an overall view of the data in some orderly manner.

If you study the Employees table in Figure 2–9 closely, you'll soon realize that there is a many-to-many relationship between the employees and committees to which they belong. A single employee can belong to any number of committees, and a single committee can be composed of any number of employees. You can, therefore, resolve these duplicate fields in the same manner that you would resolve any other many-to-many relationship—by creating a linking table. In the case of the Employees table, create the linking table by using a copy of the primary key (EmployeeID) and a single Committee field. Give the new table an appropriate name, such as Committee_Members, designate both the EmployeeID and Committee fields as a composite primary key, remove the Committee fields from the Employees table, and you're done. (You'll learn more about primary keys later in this chapter.) Figure 2–10 shows the revised Employees table and the new Committee_Members table.

Employees

EmployeeID	EmpLastName	EmpFirstName	EmpCity	*<<other fields>>*
7004	Gehring	Darren	Chico	...
7005	Kennedy	John	Portland	...
7006	Thompson	Sarah	Lubbock	...
7007	Wilson	Jim	Salem	...
7008	Seidel	Manuela	Medford	...
7009	Smith	David	Fremont	...
7010	Patterson	Neil	San Diego	...
7011	Viescas	Michael	Redmond	...

Committee_Members

EmployeeID	Committee
7004	Steering
7005	ISO 9000
7005	Safety
7006	Safety
7006	ISO 9000
7006	Steering
7008	ISO 9000
7009	Steering

Figure 2–10 *The revised Employees table and the new Committee_Members table*

Although you've resolved the duplicate fields that were in the original Employees table, you're not quite finished yet. Keeping in mind that there is a many-to-many relationship between the employees and the committees to which they belong, you might very well ask, "Where is the Committees table?" There isn't one—yet! Chances are that a committee has some other characteristics that you need to record, such as the name of the room where the committee meets and the day of the month that the meeting is held. So, you should create a real Committees table that includes fields such as CommitteeID, CommitteeName, MeetingRoom, and MeetingDay. When you finish creating the new table, replace the Committee field in the

Committee_Members table with the CommitteeID field from the new Committees table. The final structures appear in Figure 2–11.

Employees

EmployeeID	EmpLastName	EmpFirstName	EmpCity	*<<other fields>>*
7004	Gehring	Darren	Chico	...
7005	Kennedy	John	Portland	...
7006	Thompson	Sarah	Lubbock	...
7007	Wilson	Jim	Salem	...
7008	Seidel	Manuela	Medford	...
7009	Smith	David	Fremont	...
7010	Patterson	Neil	San Diego	...
7011	Viescas	Michael	Redmond	...

Committee_Members

EmployeeID	CommitteeID
7004	103
7005	104
7005	102
7006	102
7006	104
7006	103
7008	104
7009	103

Committees

CommitteeID	CommitteeName	MeetingRoom	MeetingDay
100	Budget	11-C	Tuesday
101	Christmas	9-F	Monday
102	Safety	12-B	Monday
103	Steering	12-D	Tuesday
104	ISO 9000	Main-South	Wednesday

Figure 2–11 *The final Employees, Committee_Members, and Committees structures*

You gain a real advantage by structuring the tables in this manner because you can now associate a single member with any number of committees or a single committee with any number of employees. You can then use an SQL query to view information from all three tables simultaneously.

You're now close to completing the process of fine-tuning your table structures. The last order of business is to make certain that each record within a table can be uniquely identified and that the table itself can be identified throughout the entire database.

Identification Is the Key

As you learned in Chapter 1, the primary key is one of the most important keys in a table because it uniquely identifies each record within a table and officially identifies that table throughout the database. It also establishes a relationship between a pair of tables. You cannot underestimate the importance of the primary key—every table in your database must have one!

By definition, a primary key is a field or group of fields that uniquely identifies each record within a table. A primary key is known as a *simple primary key* (or just primary key for short) when it is composed of a single field. A primary key is known as a *composite primary key* when it is composed of two or more fields. Define a simple primary key when you can because it's more efficient and is much easier to use when establishing a table relationship. Use a composite primary key only when it's appropriate (for example, to define and create a linking table).

You can use an existing field or a combination of fields as the primary key as long as they satisfy all the criteria on the following checklist. When the field or fields that you want to use as the primary key do not conform to *all* the criteria, use a different field or define a new field to act as the primary key for the table. Take some time now and use the following checklist to determine whether each primary key in your database is sound.

- *Do the fields uniquely identify each record in the table?* Each record in a table represents an instance of the subject of the table. A good primary key ensures that you have a means of accurately identifying or referencing each record in this table from other tables in the database. It also helps you to avoid having duplicate records within the table.
- *Does this field or combination of fields contain unique values?* As long as the values of the primary key are unique, you have a means of ensuring that there are no duplicate records in the table.
- *Will these fields ever contain unknown values?* This is a very important question because a primary key cannot contain unknown values. If you think this field has even the slightest possibility of containing unknown values, you should disqualify it immediately.
- *Can the value of these fields ever be optional?* If the answer to this question is "Yes," you cannot use the field in the primary key. If the value of the field can be optional, it implies that it might be unknown at some point. As you learned in the previous item, a primary key cannot contain unknown values.

- *Is this a multipart field?* Although you should have eliminated all your multipart fields by now, you should ask yourself this question anyway. If you missed a multipart field earlier, resolve it now and try to use another field as the primary key, or use the new separate fields together as a composite primary key.
- *Can the value of these fields ever be modified?* The values of primary key fields should remain static. That is, you should never change the value of a field in a primary key unless you have a truly compelling reason to do so. When the value of the field is subject to arbitrary changes, it is difficult for the field to remain in conformance with the other points in this checklist.

As we stated earlier, a field or combination of fields must pass all the points on this checklist with flying colors before it can be used as a primary key. In Figure 2–12, PilotID serves as the primary key of the Pilots table. But the question is this: Does PilotID conform to all the points on the previous checklist? If it does, the primary key is sound. But if it doesn't, you must either modify it to conform to all the points on the checklist or select a different field as the primary key.

Pilots

PilotID	PilotFirstName	PilotLastName	HireDate	Position	PilotAreaCode	PilotPhone
25100	Sam	Alborous	1994-07-11	Captain	206	555-3982
25101	Jim	Wilson	1994-05-01	Captain	206	555-6657
25102	David	Smith	1994-09-11	FirstOfficer	915	555-1992
25103	Kathryn	Patterson	1994-07-11	Navigator	972	555-8832
25104	Michael	Hernandez	1994-05-01	Navigator	360	555-9901
25105	Kendra	Bonnicksen	1994-09-11	Captain	206	555-1106

Figure 2–12 *Is PilotID a sound primary key?*

As a matter of fact, PilotID is a sound primary key because it does conform to all the points on the checklist. But what happens when you don't have a field that can act as a primary key? Take the Employees table in Figure 2–13, for example. Is there a field in this table that can act as a primary key?

It's very clear that this table doesn't contain a field (or group of fields) that can be used as a primary key. With the exception of EmpPhone, every field contains duplicate values. EmpZip, EmpAreaCode, and EmpPhone all contain unknown values. Although you might be tempted to use the combination of EmpLastName and EmpFirstName, there's no guarantee that you won't

Employees

EmpLastName	EmpFirstName	EmpCity	EmpState	EmpZip	EmpAreaCode	EmpPhone	HireDate
Gehring	Darren	Chico	CA	95926			1998-12-31
Kennedy	John	Portland	OR	97208	503	555-2621	1998-05-01
Thompson	Sarah	Redmond	WA	98052	425	555-2626	1998-09-11
Wilson	Jim	Salem	OR				1998-12-27
Seidel	Manuela	Medford	OR	97501	541	555-2641	1998-05-01
Smith	David	Fremont	CA	94538	510	555-2646	1998-09-11
Patterson	Neil	San Diego	CA	92199	619	555-2541	1998-05-01
Viescas	Michael	Redmond	WA	98052	425	555-2511	1998-09-11
Viescas	David	Portland	OR	97207	503	555-2633	1998-10-15

Figure 2–13 *Does this table have a primary key?*

employ a new person who is also named Jim Wilson or David Smith. Also, because the value of every field in this table is subject to arbitrary change, it's evident that there is no field you can use as the primary key for this table.

What do you do now? You create an artificial primary key. This is an arbitrary field you define and add to the table for the sole purpose of using it as the table's primary key. The advantage of adding this arbitrary field is that you can ensure that it conforms to all the points on the checklist. After you've added the field to the table, designate it as the primary key, and you're done! That's all there is to it. Figure 2–14 shows the Employees table with an artificial primary key called EmployeeID.

Employees

EmployeeID	EmpLastName	EmpFirstName	EmpCity	EmpState	EmpZip	*<<other fields>>*
98001	Gehring	Darren	Chico	CA	95926	...
98002	Kennedy	John	Portland	OR	97208	...
98003	Thompson	Sarah	Redmond	WA	98052	...
98004	Wilson	Jim	Salem	OR		...
98005	Seidel	Manuela	Medford	OR	97501	...
98006	Smith	David	Fremont	CA	94538	...
98007	Patterson	Neil	SanDiego	CA	92199	...
98008	Viescas	Michael	Redmond	WA	98052	...
98009	Viescas	David	Portland	OR	97207	...

Figure 2–14 *The Employees table with the new artificial primary key*

❖ **Note** Although artificial primary keys are an easy way to solve the problem, they don't really guarantee that you won't get duplicate data in your table. For example, if someone adds a new record for a person named John Kennedy and provides a new unique artificial EmployeeID value, how do you know that this second John Kennedy isn't the same as the employee 98002 already in the table?

The answer is to add a verification to your application code that checks for a potentially duplicate name and warns the user. In many database systems, you can write such validation code as something called a trigger that your database system automatically runs each time a row is changed, added, or deleted. However, discussing triggers is far beyond the scope of this book. Consult your database system documentation for details.

At this point, you've done everything you can to strengthen and fine-tune your table structures. Now we'll take a look at how you can ensure that all your table relationships are sound.

Establishing Solid Relationships

In Chapter 1, you learned that a relationship exists between a pair of tables if records in the first table are in some way associated with records in the second table. You also learned that the relationship itself can be designated as one of three types: one-to-one, one-to-many, and many-to-many. And you learned that each type of relationship is established in a specific manner. Let's review this for a moment.

❖ **Note** The diagram symbols shown in this section are part of the diagramming method presented in Mike Hernandez's book *Database Design for Mere Mortals* (Addison-Wesley, 2004). PK indicates a primary key field. FK indicates a foreign key field. CPK indicates a field that is part of a composite primary key.

- You establish a **one-to-one relationship** by taking the primary key from the primary table and inserting it into the subordinate table, where it becomes a foreign key. This is a special type of relationship because in

many cases the foreign key will also act as the primary key of the subordinate table. Figure 2–15 shows how to diagram this relationship.

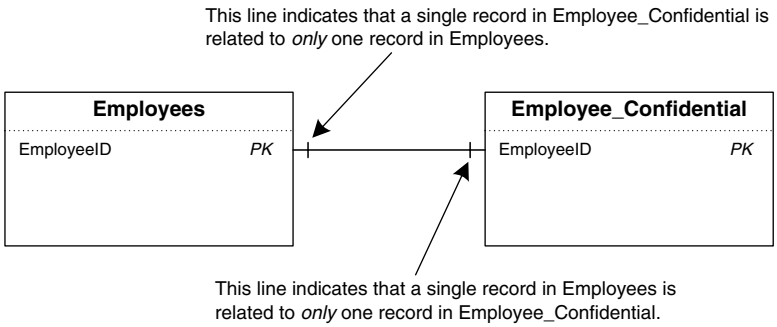

This line indicates that a single record in Employee_Confidential is related to *only* one record in Employees.
Employees
EmployeeID PK
Employee_Confidential
EmployeeID PK
This line indicates that a single record in Employees is related to *only* one record in Employee_Confidential.

Figure 2–15 *Diagramming a one-to-one relationship*

- You establish a **one-to-many relationship** by taking the primary key of the table on the "one" side and inserting it into the table on the "many" side, where it becomes a foreign key. Figure 2–16 shows how to diagram this type of relationship.

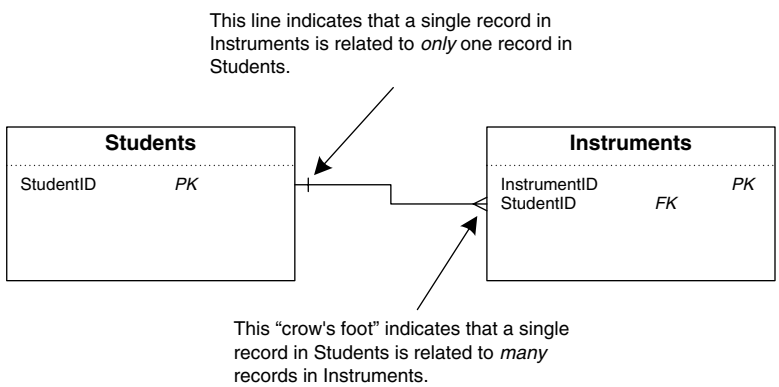

This line indicates that a single record in Instruments is related to *only* one record in Students.
Students
StudentID PK
Instruments
InstrumentID PK
StudentID FK
This "crow's foot" indicates that a single record in Students is related to *many* records in Instruments.

Figure 2–16 *Diagramming a one-to-many relationship*

- You establish a **many-to-many relationship** by creating a linking table. Define the linking table by taking a copy of the primary key of each table in the relationship and using them to form the structure of the new table. These fields commonly serve two distinct roles: Together, they form the composite primary key of the linking table; separately, they each serve as a foreign key. You would diagram this relationship as shown in Figure 2–17 (see page 44).

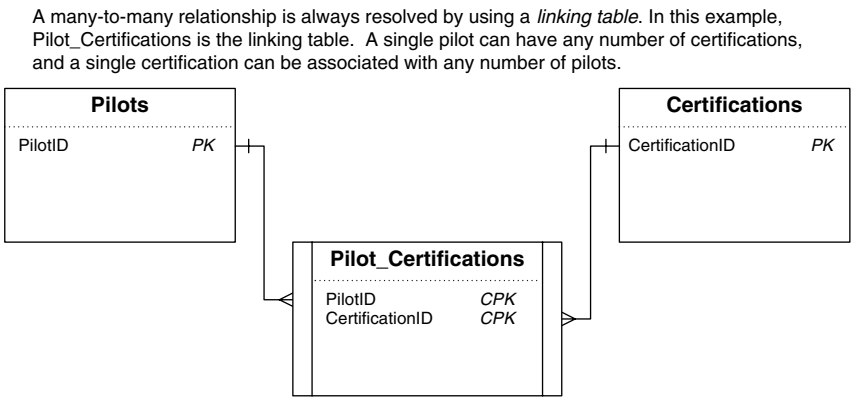

A many-to-many relationship is always resolved by using a linking table. In this example, Pilot_Certifications is the linking table. A single pilot can have any number of certifications, and a single certification can be associated with any number of pilots.
Pilots
PilotID PK
Certifications
CertificationID PK
Pilot_Certifications
PilotID CPK
CertificationID CPK

Figure 2–17 *Diagramming a many-to-many relationship*

In order to make certain that the relationships among the tables in your database are really solid, you must establish relationship characteristics for each relationship. The characteristics you're about to define indicate what will occur when you delete a record, the type of participation a table has within the relationship, and to what degree each table participates within the relationship.

Before our discussion on relationship characteristics begins, we must make one point perfectly clear: We present the following characteristics within a generic and logical frame of reference. These characteristics are important because they allow you to enforce relationship integrity (referred to by some database systems as *referential integrity*). However, the manner in which you implement them will vary from one database software program to another. You will have to study your database software's documentation to determine whether these characteristics are supported and, if so, how you can implement them.

Establishing a Deletion Rule

A *deletion rule* dictates what happens when a user makes a request to delete a record in the primary table of a one-to-one relationship or in the table on the "one" side of a one-to-many relationship. You can guard against orphaned records by establishing this rule. (*Orphaned records* are those records in the subordinate table of a one-to-one relationship that don't have related records in the primary table, or records in the table on the "many" side of a one-to-

many relationship that don't have related records in the table on the "one" side.)

You can set two types of deletion rules for a relationship: *restrict* and *cascade.*

- The **restrict deletion rule** does not allow you to delete the requested record when there are related records in the subordinate table of a one-to-one relationship or in the table on the "many" side of a one-to-many relationship. You must delete any related records *prior* to deleting the requested record. You'll use this type of deletion rule as a matter of course. In database systems that allow you to define relationship rules, this is usually the default and sometimes the only option.
- When the **cascade deletion rule** is in force, deleting the record on the "one" side of a relationship causes the system to automatically delete any related records in the subordinate table of a one-to-one relationship or in the table on the "many" side of a one-to-many relationship. Use this rule very judiciously, or you might wind up deleting records you really wanted to keep! Not all database systems support cascade deletion.

Regardless of the type of deletion rule you use, always examine your relationship very carefully in order to determine which type of rule is appropriate. You can use a very simple question to help you decide which type of rule to use. First, select a pair of tables, and then ask yourself the following question: "If a record in [name of primary or "one" side table] is deleted, should related records in [name of subordinate or "many" side table] be deleted as well?"

This question is framed in a generic sense so that you can understand the premise behind it. To apply this question, substitute the phrases within the square brackets with table names. Your question will look something like this: "If a record in the Committees table is deleted, should related records in the Committee_Members table be deleted as well?"

Use a restrict deletion rule if the answer to this question is "No." Otherwise, use the cascade deletion rule. In the end, the answer to this question greatly depends on how you use the data stored within the database. This is why you must study the relationship carefully and make certain you choose the right rule. Figure 2–18 shows how to diagram the deletion rule for this relationship. Note that you'll use (R) for a restricted deletion rule and (C) for a cascade deletion rule.

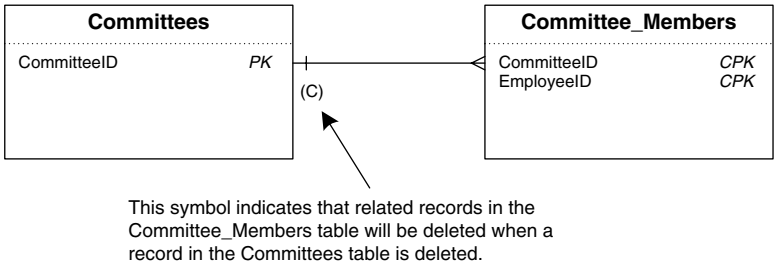

Committees
CommitteeID PK
Committee_Members
CommitteeID CPK
EmployeeID CPK
(C)
This symbol indicates that related records in the Committee_Members table will be deleted when a record in the Committees table is deleted.

Figure 2–18 *Diagramming the deletion rule for the Committees and Committee_Members tables*

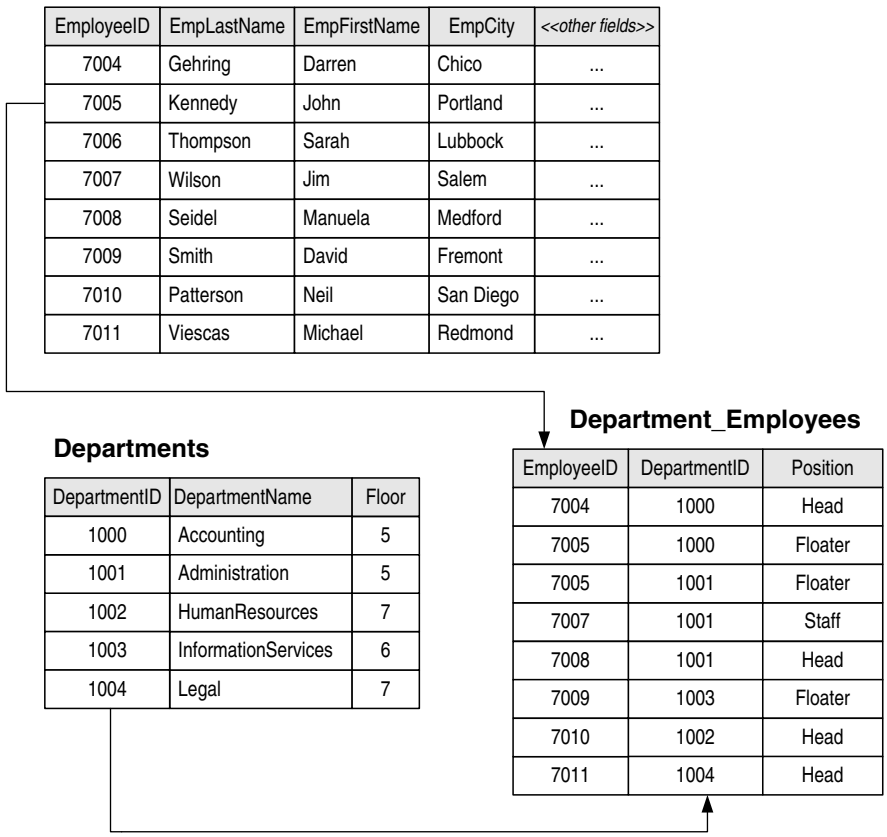
Setting the Type of Participation

When you establish a relationship between a pair of tables, each table participates in a particular manner. The *type of participation* assigned to a given table determines whether a record must exist in that table before you can enter a record into the other table. There are two types of participation.

- **Mandatory**—At least one record must exist in this table before you can enter any records into the other table.
- **Optional**—There is no requirement for any records to exist in this table before you enter any records in the other table.

The type of participation you select for a pair of tables depends mostly on the business logic of your organization. For example, let's assume you work for a large company consisting of several departments. Let's also assume that you have an Employees table, a Departments table, and a Department_Employees table in the database you've created for your company. All relevant information about an employee is in the Employees table, and all relevant information about a department is in the Departments table. The Department_Employees table is a linking table that allows you to associate any number of departments with a given employee. Figure 2–19 shows these tables. (In this figure, we used simple arrows pointing to the "many" side of the relationship.)

In the last staff meeting, you were told to assign some of the staff to a new Research and Development department. Now here's the problem: You want to make certain you add the new department to the Departments table so that you can assign staff to that department in the Department_Employees



EmployeeID	EmpLastName	EmpFirstName	EmpCity	<<other fields>>
7004	Gehring	Darren	Chico	...
7005	Kennedy	John	Portland	...
7006	Thompson	Sarah	Lubbock	...
7007	Wilson	Jim	Salem	...
7008	Seidel	Manuela	Medford	...
7009	Smith	David	Fremont	...
7010	Patterson	Neil	San Diego	...
7011	Viescas	Michael	Redmond	...

Departments

DepartmentID	DepartmentName	Floor
1000	Accounting	5
1001	Administration	5
1002	HumanResources	7
1003	InformationServices	6
1004	Legal	7

Department_Employees

EmployeeID	DepartmentID	Position
7004	1000	Head
7005	1000	Floater
7005	1001	Floater
7007	1001	Staff
7008	1001	Head
7009	1003	Floater
7010	1002	Head
7011	1004	Head

Figure 2–19 *The Employees, Departments, and Department_Employees tables*

table. This is where the type of participation characteristic comes into play. Set the type of participation for the Departments table to mandatory and the type of participation for the Department_Employees table to optional. By establishing these settings, you ensure that a department must exist in the Departments table before you can assign any employees to that department in the Department_Employees table.

As with the deletion rule, study each relationship carefully to determine the appropriate type of participation setting for each table in the relationship. You would diagram the type of participation as shown in Figure 2–20 (see page 48).

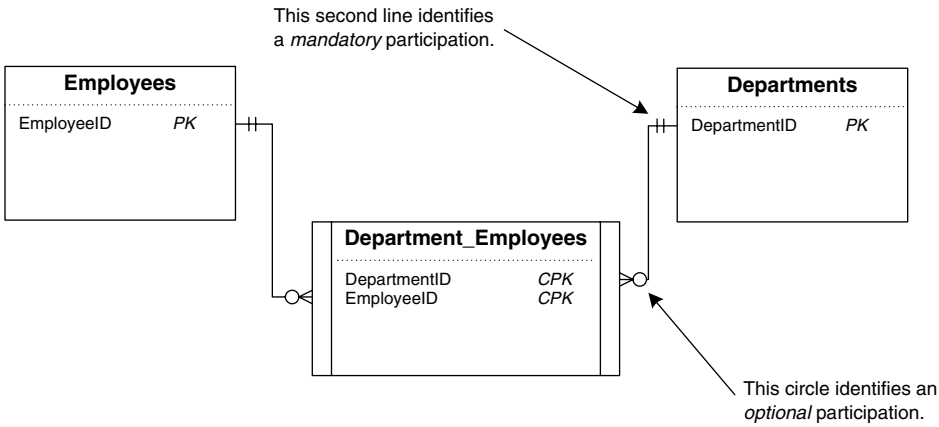


Figure 2–20 *Diagramming the type of participation for the Departments and Department_Employees tables*

Setting the Degree of Participation

Now that you've determined *how* each table will participate in the relationship, you must figure out *to what degree* each will participate. You do this by determining the minimum and maximum number of records in one table that can be related to a single record in the other table. This process is known as identifying a table's *degree of participation*. The degree of participation for a given table is represented by two numbers that are separated with a comma and enclosed within parentheses. The first number indicates the minimum possible number of related records, and the second number indicates the maximum possible number of related records. For example, a degree of participation such as "(1,12)" indicates that the minimum number of records that can be related is one and the maximum is twelve.

The degree of participation you select for various tables in your database largely depends on how your organization views and uses the data. Let's say that you're a booking agent for a talent agency and that two of the tables in your database are Agents and Entertainers. Let's further assume that there is a one-to-many relationship between these tables—one record in the Agents table can be related to many records in the Entertainers table, but a single record in the Entertainers table can be related to only one record in the Agents table. In this case, we've ensured (in a general sense) that an entertainer is assigned to only one agent. (We definitely avoid the possibility of the entertainer playing one agent against another. This is a good thing.)

In nearly all cases, the maximum number of records on the "many" side of a relationship will be infinite. However, in some cases your business rules might dictate that you limit this participation. One example would be to limit the number of students who can enroll in a class. In this example, let's assume that the boss wants to ensure that all his agents have a fair shake at making good commissions and wants to keep the infighting between agents down to a bare minimum. So he sets a new policy stating that a single agent can represent a maximum of six entertainers. (Although he thinks it might not work in the long run, he wants to try it anyway.) In order to implement his new policy, he sets the degree of participation for both tables to the following:

Agents	(1,1)—An entertainer can be associated with one and only one agent.
Entertainers	(0,6)—Although an agent doesn't have to be associated with an entertainer at all, he or she cannot be associated with more than six entertainers at any given time.

Figure 2–21 shows how to diagram the degree of participation for these tables.

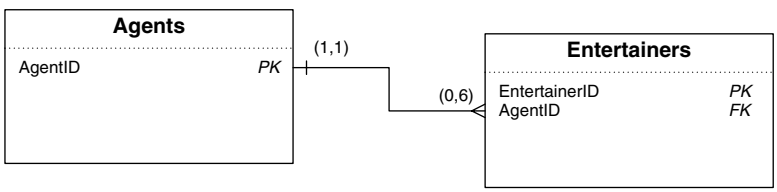


Figure 2–21 *Diagramming the degree of participation for the Agents and Entertainers tables*

After setting the degree of participation, you should decide how you want your database system to enforce the relationship. What you choose depends on the features provided by your database system. The simplest enforcement supported by most database systems is to restrict the values in the foreign key in the "many" table so that the user cannot enter a value that is not in the related "one" table. You can indicate this by placing the letter R in parentheses next to the relationship line pointing to the "one" table, as shown in Figure 2–22 (see page 50).

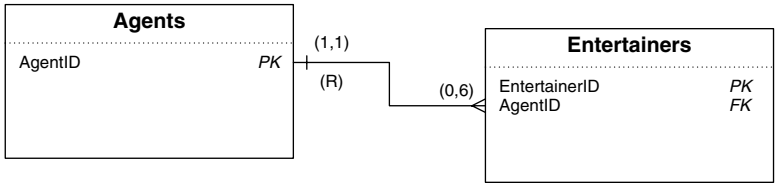


Figure 2–22 *A diagram of all the relationship characteristics for the Agents and Entertainers tables*

Some database systems allow you to define a rule that cascades (C) the key value from the "one" table to the "many" table if the user changes the value of the primary key in the "one" table. Essentially, the database system corrects the foreign key value in related rows in the "many" table when you change the value of the primary key in the "one" table. And some database systems provide a feature that automatically deletes (D) the rows in the "many" table when you delete a row in the "one" table. Check your database system documentation for details.

❖ **Note** To actually enforce degree of participation constraints, you'll have to define one or more triggers or constraints in your database definition (if your database system supports these features).

Is That All?

By using the techniques you learned in this chapter, you make the necessary beginning steps toward ensuring a fundamental level of data integrity in your database. The next step is to begin studying the manner in which your organization views and uses its data so that you can establish and impose business rules for your database. But to really get the most from your database, you should go back to the beginning and run it through a thorough database design process using a good design methodology. Unfortunately, these topics are beyond the scope of this book. However, you can learn a good design methodology from books such as *Database Design for Mere Mortals* (Addison-Wesley, 2004) by Michael J. Hernandez or *Database Systems: A Practical Approach to Design, Implementation, and Management,* fourth edition (Addison-Wesley, 2004) by Thomas Connolly and Carolyn Begg. The point to remember is this: The more solid your database structure, the easier it will be

both to extract information from the data in the database and to build applications programs for it.

SUMMARY

We opened this chapter with a short discussion on why you should be concerned with having sound structures in your database. You learned that poorly designed tables can cause numerous problems, not the least of which concern data integrity.

Next we discussed fine-tuning the fields in each table. You learned that giving your fields good names is very important because it ensures that each name is meaningful and actually helps you to find hidden problems with the field structure itself. You now know how to fine-tune your field structures by ensuring they conform to a few simple rules. These rules deal with issues such as guaranteeing that each field represents a single characteristic of the table's subject, contains only a single value, and never stores a calculation. We also discussed the problems found in multipart and multivalued fields, and you learned how to resolve them properly.

Fine-tuning the tables was the next issue we addressed. You learned that the table names are just as important an issue as field names for many of the same reasons. You now know how to give your tables meaningful names and ensure that each table represents only a single subject. We then discussed a set of rules you can use to make certain each table structure is sound. Although some of the rules seemed to duplicate some of the efforts you made in fine-tuning your field structures, you learned that the rules used for fine-tuning the table structures actually add an extra level of insurance in making sure that the table structures are as absolutely sound as they can be.

The next subject we tackled was primary keys. You learned the importance of establishing a primary key for each table in your database. You now know that a primary key must conform to a specific set of characteristics and that the field that will act as the primary key of a table must be chosen very carefully. You also learned that you can create an artificial primary key if there is no field in the table that conforms to the complete set of characteristics for a primary key.

We closed this chapter with a discussion on establishing solid relationships. After reviewing the three types of relationships, you learned how to diagram each one. You then learned how to establish and diagram a deletion rule for

the relationship. This rule is important because it helps you guard against orphaned records. The last two topics we discussed were the type of participation and degree of participation for each table within the relationship. You learned that a table's participation can be mandatory or optional and that you can set a specific range for the number of related records between each table.

In the next chapter, you'll learn a little bit about the history of SQL and how it evolved into its current version, SQL:2003.

3

A Concise History of SQL

"There is only one religion, though there are many versions of it."
—George Bernard Shaw
Plays Pleasant and Unpleasant

Topics Covered in This Chapter

The Origins of SQL
Early Vendor Implementations
". . . And Then There Was a Standard"
Evolution of the ANSI/ISO Standard
Commercial Implementations
What the Future Holds
Why Should You Learn SQL?
Summary

The telling of history always involves vague and ambiguous accounts of various incidents, political intrigue, and human foibles. The history of SQL is no different than that of any other subject in this sense. SQL has been around in one form or another since just after the dawn of the relational model, and there are several detailed accounts of its long and spotty existence. In this chapter, however, we take a close look at the origin, evolution, and future of this database language. We have two goals: first, to give you an idea of how SQL matured into the language used by a majority of relational database systems today, and second, to give you a sense of why it is important for you to learn how to use SQL.

The Origins of SQL

As you learned in Chapter 1, Dr. E. F. Codd presented the relational database model to the world in 1970. Soon after this landmark moment, organizations such as universities and research laboratories began efforts to develop a language that could be used as the foundation to a database system that supported the relational model. Initial work led to the development of several languages in the mid- to early 1970s, and later efforts resulted in the development of SQL and the SQL-based databases in use today. But just where did SQL originate? How did it evolve? What is its future? For the answers to these questions, we must begin our story at IBM's Santa Teresa Research Laboratory in San Jose, California.

IBM began a major research project in the early 1970s called System/R. The goals of this project were to prove the viability of the relational model and to gain some experience in designing and implementing a relational database. The researchers' initial endeavors between 1974 and 1975 proved successful, and they managed to produce a minimal prototype of a relational database.

In addition to their efforts to develop a working relational database, researchers were also working to define a database language. The work performed at this laboratory is arguably the most commercially significant of the initial efforts to define such a language. In 1974, Dr. Donald Chamberlin and his colleagues developed Structured English Query Language (SEQUEL). The language allowed users to query a relational database using clearly defined English-style sentences. Dr. Chamberlin and his staff first implemented this new language in a prototype database called SEQUEL-XRM.

The initial feedback and success of SEQUEL-XRM encouraged Dr. Chamberlin and his staff to continue their research. They completely revised SEQUEL between 1976 and 1977 and named the new version SEQUEL/2. However, they subsequently had to change the name SEQUEL to SQL (Structured Query Language or SQL Query Language) for legal reasons—someone else had already used the acronym SEQUEL. To this day, many people still pronounce SQL as *sequel*, although the widely accepted "official" pronunciation is *es-cue-el*. SQL provided several new features, such as support for multi-table queries and shared data access by multiple users.

Soon after the emergence of SQL, IBM began a new and more ambitious project aimed at producing a prototype database that would further substantiate the feasibility of the relational model. They called the new prototype

System R and based it on a large subset of SQL. After much of the initial development work was completed, IBM installed System R in a number of internal sites and selected client sites for testing and evaluation. Many changes were made to System R and SQL based on the experiences and feedback of users at these sites. IBM closed the project in 1979 and concluded that the relational model was indeed a viable database technology with commercial potential.

❖ **Note** One of the more important successes attributed to this project is the development of SQL. But SQL's roots are actually based in a research language called Specifying Queries As Relational Expressions (SQUARE). This language was developed in 1975 (predating the System R project) and was designed to implement relational algebra with English-style sentences.

You might well ask, "If IBM concluded that there was commercial potential, why did the company close the project?" John remembers seeing a demonstration of System R in the late 1970s. It had lots of "wow" factor, but on the hardware technology available at the time, even a simple query took minutes to run. It clearly had potential, but it definitely needed better hardware and software to make the product appealing to businesses.

Early Vendor Implementations

The work done at the IBM research lab during the 1970s was followed with great interest in various technical journals, and the merits of the new relational model were briskly debated at database technology seminars. Toward the latter part of the decade, it became clear that IBM was keenly interested in and committed to developing products based on relational database technology and SQL. This, of course, led many vendors to speculate how soon IBM would roll out its first product. Some vendors had the good sense to start work on their own products as quickly as possible and not wait around for IBM to lead the market.

In 1977, Relational Software, Inc. was formed by a group of engineers in Menlo Park, California, for the purpose of building a new relational database product based on SQL. They called their product Oracle. Relational Software shipped its product in 1979, beating IBM's first product to market by two years and providing the first commercially available relational database man-

agement system (RDBMS). One of Oracle's advantages was that it ran on Digital's VAX minicomputers instead of the more expensive IBM mainframes. Relational Software has since been renamed to Oracle Corporation and is one of the leading vendors of RDBMS software.

Meanwhile, Michael Stonebraker, Eugene Wong, and several other professors at the University of California's Berkeley computer laboratories were also researching relational database technology. Like the IBM team, they developed a prototype relational database and dubbed their product INGRES. INGRES included a database language called Query Language (QUEL), which, in comparison to SQL, was much more structured but made less use of English-like statements. INGRES was eventually converted to an SQL-based RDBMS when it became clear that SQL was emerging as the standard database language. Several professors left Berkeley in 1980 to form Relational Technology, Inc., and in 1981 they announced the first commercial version of INGRES. Relational Technology has gone through several transformations and is now part of Computer Associates International, Inc. INGRES is still one of the leading database products in the industry today.

Now we come full circle back to IBM. IBM announced its own RDBMS called SQL/Data System (SQL/DS) in 1981 and began shipping it in 1982. In 1983, the company introduced a new version of SQL/DS for the VM/CMS operating system (one of several offered by IBM for their mainframe systems) and announced a new RDBMS product called Database 2 (DB2), which could be used on IBM mainframes using IBM's mainstream MVS operating system. First shipped in 1985, DB2 has become IBM's premiere RDBMS, and its technology has been incorporated into the entire IBM product line. By the way, IBM hasn't changed—it's still IBM.

During the course of more than 25 years, we've seen what began as research for the System R project become a force that impacts almost every level of business today and evolve into a multibillion dollar industry.

". . . And Then There Was a Standard"

With the flurry of activity surrounding the development of database languages, you could easily wonder if anyone ever thought of standardization. Although the idea was tossed about among the database community, there was never any consensus or agreement as to who should set the standard or which dialect it should be based upon. So each vendor continued to develop

and improve its own database product in the hope that it—and by extension, its dialect of SQL—would become the industry standard.

Customer feedback and demand drove many vendors to include certain elements in their SQL dialects, and in time an unofficial standard emerged. It was a small specification by today's standards, as it encompassed only those elements that were similar across the various SQL dialects. However, this specification (such as it was) did provide database customers with a core set of criteria by which to judge the various database programs on the market, and it also gave users a small set of knowledge that they could leverage from one database program to another.

In 1982, the American National Standards Institute (ANSI) responded to the growing need for an official relational database language standard by commissioning its X3 organization's database technical committee, X3H2, to develop a proposal for such a standard. (X3 is one of many organizations overseen by ANSI.) In turn, X3H2 is only one of many technical committees that report to X3. X3H2 was and continues to be composed of database industry experts and representatives from almost every major SQL-based database vendor. In the beginning, the committee reviewed and debated the advantages and disadvantages of various proposed languages and also began work on a standard based on QUEL, the database language for INGRES. But market forces and the increasing commitment to SQL by IBM induced the committee to base its proposal on SQL instead.

The X3H2 committee's proposed standard was largely based on IBM's DB2 SQL dialect. The committee worked on several versions of its standard over the next two years and even improved SQL to some extent. However, an unfortunate circumstance arose as a result of these improvements: This new standard became incompatible with existing major SQL dialects. X3H2 soon realized that the changes made to SQL did not significantly improve it enough to warrant the incompatibilities, so the committee reverted to the original version of the standard.

ANSI ratified X3H2's standard in 1986 as "ANSI X3.135-1986 Database Language SQL," which became commonly known as SQL/86. Although X3H2 made some minor revisions to its standard before it was adopted by ANSI, SQL/86 merely defined a minimal set of "least common denominator" requirements to which database vendors could conform. In essence, it conferred official status on the elements that were similar among the various SQL dialects and that had already been implemented by many database vendors. But the

new standard finally provided a specific foundation from which the language and its implementations could be developed further.

The International Organization for Standardization (ISO) approved its own document (which corresponded exactly with ANSI SQL/86) as an international standard in 1987 and published it as "ISO 9075-1987 Database Language SQL." (Both standards are still often referred to as just SQL/86.) The international database vendor community could now work from the same standards as those vendors in the United States. Despite the fact that SQL gained the status of an official standard, the language was far from being complete.

Evolution of the ANSI/ISO Standard

SQL/86 was soon criticized in public reviews, by the government, and by industry pundits such as C. J. Date. Some of the problems cited by these critics included redundancy within the SQL syntax (there were several ways to define the same query), lack of support for certain relational operators, and lack of referential integrity. Although X3H2 knew of these problems even before SQL/86 was published, the committee decided that it was better to release a standard now (even though it still needed work) than to have no standard at all.

Both ISO and ANSI addressed the criticism pertaining to referential integrity by adopting refined versions of their standards. ISO published "ISO 9075:1989 Database Language SQL with Integrity Enhancements" in mid-1989, while ANSI adopted its "X3.135-1989 Database Language SQL with Integrity Enhancements," also often referred to as SQL/89, late that same year. But the ANSI committee's work for the year wasn't over just yet. X3H2 was still trying to address an important issue brought forth by the government.

Some government users complained that the specification explaining how to embed SQL within a conventional programming language was not an explicit component of the standard. (Although the specification was included, it was relegated to an appendix.) Their concern was that vendors might not support portable implementations of embedded SQL because there was no specific requirement within the standard for them to do so. X3H2 responded by developing a second standard that required conformance to the embedding specification, publishing it as "ANSI X3.168-1989 Database Language Embedded SQL." It's interesting to note that ISO chose not to publish a corresponding standard because of a lack of similar concern within the international

community. This meant that ISO had no specification for embedding SQL within a programming language, a situation that would not change until ISO's publication of its SQL/92 Standard.

SQL/86 and SQL/89 were far from being complete standards—they lacked some of the most fundamental features needed for commercial database systems. For example, neither standard specified a way to make changes to the database structure (including within the database system itself) after it was defined. No one could modify or delete any structural components (such as tables or columns) or make any changes to the security of the database. For example, you could CREATE a table, but the standard included no definition of the DROP command to delete a table or the ALTER command to change it. Also, you could GRANT security access to a table, but the standard did not define the REVOKE command to allow removal of access authority. Ironically, these capabilities were provided by all commercial SQL-based databases. They were not included in either standard, however, because each vendor implemented them in different ways. Other features were widely implemented among many SQL-based databases but omitted from the standards. Once again, it was an issue of varied implementations.

By the time SQL/89 was completed, both ANSI and ISO were already working on major revisions to SQL that would make it a complete and robust language. The new version would be referred to as SQL/92 (what else?) and would include features that had already been widely implemented by most major database vendors. But one of the main objectives of both ANSI and ISO was to avoid defining a "least common denominator" standard yet again. As a result, they decided to both include features that had not yet gained wide acceptance and add new features that were substantially beyond those currently implemented.

ANSI and ISO published their new SQL Standards—"X3.135-1992 Database Language SQL" and "ISO/IEC 9075:1992 Database Language SQL," respectively—in October 1992. (Work on these documents was completed in late 1991, but some final fine-tuning took place during 1992.) The SQL/92 document is considerably larger than the one for SQL/89, but it's also much broader in scope. For example, it provides the means to modify the database structure after it has been defined, supports additional operations for manipulating character strings as well as dates and times, and defines additional security features. SQL/92 was a major step forward from any of its predecessors.

Fortunately, the standards committees anticipated this situation to some extent. In order to facilitate a smooth and gradual conformance to the new standard, ANSI and ISO defined SQL/92 on three levels.

ENTRY SQL	Similar to SQL/89, this level also includes features to make the transition from SQL/89 to SQL/92 easier as well as features that corrected errors in the SQL/89 Standard. The idea was that this level would be the easier to implement because most of its features had already been widely incorporated into existing products.
INTERMEDIATE SQL	This level encompasses most of the features in the new standard. Both committees' decisions to include certain features at this level were based on several factors. The overall objectives were to enhance the standard so that SQL better supported the concepts in the relational model and to redefine syntax that was ambiguous or unclear. It was an easy decision to include features that were already implemented in some way by one or more vendors and that met these objectives. Features demanded by users of SQL database systems were given high consideration as long as they met these objectives and were relatively easy for most vendors to implement. This level was meant to ensure that it would be reasonably possible for a given product to have as robust an implementation as possible.
FULL SQL	The entire SQL/92 specification is encompassed within this level. It obviously includes the more complex features that were omitted in the first two levels. This level includes features that, although considered important to meet customer demands or further "purify" the language, would be difficult for most vendors to implement immediately. Unfortunately, compliance with Full SQL is not yet a requirement, so it will be some time before we can expect database products to fully implement the standard.

Although many database vendors continued work on implementing the features in SQL/92, they also developed and implemented features of their own. The additions they made to the SQL Standard are known as *extensions*. For example, a vendor might provide more data types than the six specified in

SQL/92. Although these extensions provide more functionality within a given product and allow vendors to differentiate themselves from one another, there are drawbacks. The main problem with adding extensions is that it causes each vendor's dialect of SQL to diverge further from the original standard. This, in turn, prevents database developers from creating portable applications that can be run from any SQL database.

Other SQL Standards

The ANSI/ISO SQL Standard is the most widely accepted standard to date. This means, of course, that other standards in existence also incorporate SQL in one form or another. These are some of the more significant alternate standards.

X/OPEN — A group of European vendors (collectively known as X/OPEN) developed a set of standards that would help establish a portable application environment based on UNIX. The ability to port an application from one computer system to another without changing it is an important issue in the European market. Although the X/OPEN members have adopted SQL as part of this set of standards, their version deviates from the ANSI/ISO Standard in several areas.

SAA — IBM has always developed its own dialect of SQL, which the company incorporated into its Systems Application Architecture (SAA) specification. Integrating IBM's SQL dialect into the complete line of IBM database products was one of the goals of the SAA specification. Although this goal has never been achieved, SQL still plays an important role in unifying IBM's database products.

FIPS — The National Institute of Standards and Technology (NIST) made SQL a Federal Information Processing Standard (FIPS) beginning in 1987. Originally published as "FIPS PUB 127," it specifies the level to which an RDBMS must conform to the ANSI/ISO Standard. Since then, all relational database products used by the U.S. government have been required to conform to the current FIPS publication.

ODBC — In 1989 a group of database vendors formed the SQL Access group to address the problem of database interoperability. Although these vendors' first efforts were somewhat unsuccessful, they widened their focus to include a way to bind an SQL database to a user-interface language. The result of their efforts was the Call-Level Interface (CLI) specification published in

1992. That same year, Microsoft published its Open Database Connectivity (ODBC) specification, which was based on the CLI Standard. ODBC has since become the *de facto* means of accessing and sharing data among SQL databases that support it.

These standards continually evolve as newer versions of ANSI/ISO SQL are adopted, and they are sometimes independently developed as well.

In 1997, ANSI's X3 organization was renamed the National Committee for Information Technology Standards (NCITS), and the technical committee in charge of the SQL Standard is now called ANSI NCITS-H2. Because of the rapidly growing complexity of the SQL Standard, the ANSI and ISO standards committees agreed to break the standard into twelve separate numbered parts and one addendum as they began to work on SQL3 (so named because it's the third major revision of the standard) so that work on each part could proceed in parallel. Since 1997, two additional parts have been defined.

Table 3–1 shows the name and description of each part of the SQL Standard, as well as the status of each part as of early 2007.

Table 3–1 *Structure of the SQL Standard*

Name	Status	Description	Pages in SQL:2003
Part 1: SQL/Framework	Completed in 1999 and updated in 2003.	Describes each part of the standard and contains information common to all parts.	81
Part 2: SQL/Foundation	The core 1992 standard that has been updated in 1999 and 2003.	Defines the syntax and semantics of the data definition and data manipulation portions of the SQL language.	1,267
SQL/OLAP (Online Analytical Processing)	Merged with Foundation in 1999.	Describes the functions and operations used for analytical processing. (This is intended as an amendment to SQL/Foundation.)	
Part 3: SQL/CLI (Call-Level Interface)	Completed in 1995 and expanded in 1999 and 2003.	Developed by the SQL Access group, this part corresponds to Microsoft's ODBC specification.	405

Table 3–1 *Continued*

Name	Status	Description	Pages in SQL:2003
Part 4: SQL/PSM (Persistent Stored Modules)	Completed in 1996. Stored routines and the CALL statement moved to Foundation in 1999. Remaining standard updated in 2003.	Defines procedural language SQL statements that are useful in user-defined functions and procedures. (Support for stored procedures, stored functions, the CALL statement, and routine invocation was eventually moved to SQL/Foundation.)	184
Part 5: SQL/Bindings	Specification for embedding SQL moved to a separate part in 1999 and then was embedded in Foundation in 2003.	Specifies how SQL is embedded in non-object programming languages. This part will be merged into SQL/Foundation in the next version of SQL.	
Part 6: Transaction (XA Specialization)	Canceled in 1999.	SQL specialization of the X/OPEN XA specification.	
Part 7: SQL/Temporal	Withdrawn in 2003.	Defines support for storage and retrieval of temporal data. There has been some difference of opinion on the requirements and details of Temporal, so work has stalled over the last several years.	
Part 8: SQL/Objects—Extended Objects	Merged into Foundation in 1999.	Defines how application-defined abstract data types are handled by the RDBMS.	
Part 9: SQL/MED (Management of External Data)	ISO version completed in 2003.	Defines additional syntax and definitions to SQL/Foundation that allow SQL to access non-SQL data sources (files).	498

(continued)

Table 3–1 *Continued*

Name	Status	Description	Pages in SQL:2003
Part 10: SQL/OLB (Object Language Bindings)	Completed in 1998 as an ANSI-only standard, revised in 1999 by ISO, and revised again in 2003.	Specifies the syntax and semantics of embedding SQL in the Java programming language. This corresponds to another ANSI standard, SQLJ Part 0.	405
Part 11: SQL/Schemata	Extraction from Foundation completed in 2003.	Information and definition schemas.	296
Part 12: SQL/ Replication	Project started in 2000 but dropped in 2003 due to lack of progress.	Defines support and facilities for replicating an SQL database.	
Part 13: JRT—SQL Routines Using the Java Programming Language	Completed in 1999 as an ANSI-only standard based on SQL/92. Revised as an international standard in 2003.	Defines how Java code can be used within an SQL database.	204
Part 14: SQL/XML—SQL Routines Using the eXtensible Markup Language	Completed in 2003 and expanded in 2006.	Defines how XML can be used within an SQL database. This part is aligned with the W3C XQuery V1.1 specification.	266

Commercial Implementations

As you read earlier in this chapter, SQL first appeared in the mainframe environment. Products such as DB2, INGRES, and Oracle have been around since 1979 and have legitimized the use of SQL as the preferred method of working with relational databases. During the 1980s, relational databases hit the desktop on personal computers, and products such as R:BASE, dBase IV, and Super Base put the power of data in tables at the user's fingertips. However, it wasn't until the very late 1980s and early 1990s that SQL became the language of

choice for desktop relational databases. The product that arguably broke the dam was Microsoft Access version 1 in 1992.

The early 1990s also heralded the advent of client/server computing, and RDBMS programs such as Microsoft SQL Server and Informix-SE have been designed to provide database services to users in numerous types of multi-user environments. Since 2000 there has been a concerted effort to make database information available via the Internet. Businesses have caught on to the idea of e-commerce, and those who haven't already established a Web presence are moving quickly to do so. As a result, database developers are demanding more powerful client/server databases and newer versions of long-established mainframe RDBMS products that they can use to develop and maintain the databases needed for their Web sites.

We could attempt to list all the mainstream products that support SQL, but the list would go on for pages and pages. Suffice it to say that SQL in commercial database systems is here to stay.

What the Future Holds

When we first wrote this book in 1999, the standards committees were just putting the finishing touches on SQL3, which had been a long time in coming. Since then, SQL:1999 and SQL:2003 have been published. As of mid-2007, both the ANSI and ISO committees are hard at work producing revisions for SQL:2007. Extensive revisions are planned to Part 14, SQL/XML. The international committee is also working on a separate SQL/MM—Multimedia standard that has its own five parts: Framework, Full Text, Spatial, Still Image, and Data Mining. Although the standards committees started out far behind the commercial implementations in 1986, it's fair to say that the SQL Standard long ago caught up with, and in many areas is now staying ahead of, features in available database systems.

Why Should You Learn SQL?

Learning SQL gives you the skills you need to retrieve information from any relational database. It also helps you understand the mechanisms behind the graphical query interfaces found in many RDBMS products. Understanding SQL helps you craft complex queries and provides the knowledge required to troubleshoot queries when problems occur.

Because SQL is found in a wide variety of RDBMS products, you can use your skills across a variety of platforms. For example, after you learn SQL in a product such as Microsoft Access, you can leverage your existing knowledge if your company decides to move to Microsoft SQL Server, Oracle Corporation's Oracle, or IBM's DB/2. You won't have to relearn SQL—you'll just have to learn the differences between the first dialect that you learn and the dialect used in another product.

It bears repeating that SQL is here to stay. Many vendors have invested huge amounts of money, time, and research to incorporate SQL into their RDBMS products, and a vast number of businesses and organizations have built much of their information technology infrastructures on those products. As you have probably surmised by what you've learned in this chapter, SQL will continue to evolve to meet the changing demands and requirements of the marketplace.

SUMMARY

We began this chapter with a discussion on the origins of SQL. You learned that SQL is a relational database language that was created soon after the introduction of the relational model. We also explained that the early evolution of SQL was closely tied to the evolution of the relational model itself.

Next, we discussed the initial implementations of the relational model by various database vendors. You learned that the first relational databases were implemented on mainframe computers. You also learned how IBM and Oracle came to be big players in the database industry.

We then discussed the origin of the ANSI SQL Standard. You learned that there was an unofficial standard before ANSI decided to define an official one, and we discussed the ANSI X3H2 committee's initial work on the specification. We explained that although the new standard was basically a set of "least common denominator" features, it did provide a foundation from which the language could be further developed. You also learned that the ISO published its own standard, which corresponded exactly with the ANSI specification.

The evolution of the ANSI/ISO Standard was the next topic of discussion, and you learned that various people and organizations criticized the initial standards. We then discussed how ANSI/ISO responded to the criticisms by adopting several revisions to the standard. You learned how one version led to the next and how we arrived at the SQL/92 Standard. We explained how

that standard defined various conformance levels that allowed vendors to implement the standard's features into their products as smoothly as possible. Next, we discussed the progress that the SQL Standard has made since 1992, and we took a quick look at the evolution of commercial SQL databases.

We closed the chapter with a short discussion on the future of SQL. You learned that SQL:2003 is a much more complex standard than SQL/92. We also explained why SQL will continue to be developed and gave you some good reasons for learning the language.

This page intentionally left blank

Part II

SQL Basics

This page intentionally left blank

4

Creating a Simple Query

"Think like a wise man but communicate in the language of the people."
—William Butler Yeats

Topics Covered in This Chapter

Introducing SELECT

The SELECT Statement

A Quick Aside: Data versus Information

Translating Your Request into SQL

Eliminating Duplicate Rows

Sorting Information

Saving Your Work

Sample Statements

Summary

Problems for You to Solve

Now that you've learned a little bit about the history of SQL, it's time to jump right in and learn the language itself. As we mentioned in the Introduction, we're going to spend most of this book covering the data manipulation portion of the language. So our initial focus will be on the true workhorse of SQL—the SELECT statement.

Introducing SELECT

Above all other keywords, SELECT truly lies at the heart of SQL. It is the cornerstone of the most powerful and complex statement within the language and the means by which you retrieve information from the tables in your database. You use SELECT in conjunction with other keywords and clauses to find and view information in an almost limitless number of ways. Nearly any question regarding who, what, where, when, or even what if and how many can be answered with SELECT. As long as you've designed your database properly and collected the appropriate data, you can get the answers you need to make sound decisions for your organization. As you'll discover when you get to Part V, Modifying Sets of Data, you'll apply many of the techniques you learn about SELECT to create UPDATE, INSERT, and DELETE statements.

The SELECT operation in SQL can be broken down into three smaller operations, which we will refer to as the SELECT statement, the SELECT expression, and the SELECT query. (Breaking down the SELECT operation in this manner will make it far easier to understand and to appreciate its complexity.) Each of these operations provides its own set of keywords and clauses, providing you with the flexibility to create a final SQL statement that is appropriate for the question you want to pose to the database. As you'll learn in later chapters, you can even combine the operations in various ways to answer very complex questions.

In this chapter, we'll begin our discussion of the SELECT statement and take a brief look at the SELECT query. We'll then examine the SELECT statement in more detail as we work through to Chapter 5, Getting More Than Simple Columns, and Chapter 6, Filtering Your Data.

❖ **Note** In other books about relational databases, you'll sometimes see he word *relation* used for *table*, and you might encounter *tuple* or *record* for *row* and perhaps *attribute* or *field* for *column*. However, the SQL Standard specifically uses the terms *table*, *row*, and *column* to refer to these particular elements of a database structure. We'll stay consistent with the SQL Standard and use these latter three terms throughout the remainder of the book.

The SELECT Statement

The SELECT statement forms the basis of every question you pose to the database. When you create and execute a SELECT statement, you are querying the database. (We know it sounds a little obvious, but we want to make certain that everyone reading this starts from the same point of reference.) In fact, many RDBMS programs allow you to save a SELECT statement as a *query*, *view*, *function*, or *stored procedure*. Whenever someone says she is going to query the database, you know that she's going to execute some sort of SELECT statement. Depending on the RDBMS program, SELECT statements can be executed directly from a command line window, from an interactive Query by Example (QBE) grid, or from within a block of programming code. Regardless of how you choose to define and execute it, the syntax of the SELECT statement is always the same.

❖ **Note** Many database systems provide extensions to the SQL Standard to allow you to build complex programming statements (such as If . . . Then . . . Else) in functions and stored procedures, but the specific syntax is unique to each different product. It is far beyond the scope of this book to cover even one or two of these programming languages—such as Microsoft SQL Server's Transact-SQL or Oracle's PL/SQL. You'll still use the cornerstone SELECT statement when you build functions and stored procedures for your particular database system. Throughout this book, we'll use the term *view* to refer to a saved SQL statement even though you might embed your SQL statement in a function or procedure.

A SELECT statement is composed of several distinct keywords, known as *clauses*. You define a SELECT statement by using various configurations of these clauses to retrieve the information you require. Some of these clauses are required, although others are optional. Additionally, each clause has one or more keywords that represent required or optional values. These values are used by the clause to help retrieve the information requested by the SELECT statement as a whole. Figure 4–1 (on page 73) shows a diagram of the SELECT statement and its clauses.

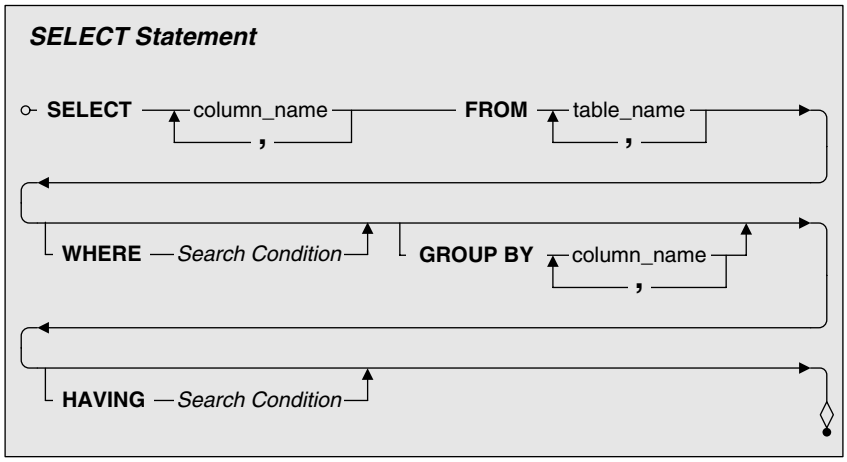


Figure 4–1 *A diagram of the SELECT statement*

❖ **Note** The syntax diagram in Figure 4–1 reflects a rudimentary SELECT statement. We'll continue to update and modify the diagram as we introduce and work with new keywords and clauses. So for those of you who might have some previous experience with SQL statements, just be patient and bear with us for the time being.

Here's a brief summary of the clauses in a SELECT statement.

- **SELECT**—This is the primary clause of the SELECT statement and is absolutely required. You use it to specify the columns you want in the result set of your query. The columns themselves are drawn from the table or view you specify in the FROM clause. (You can also draw them from several tables simultaneously, but we'll discuss this later in Part III, Working with Multiple Tables.) You can also use aggregate functions, such as Sum(HoursWorked), or mathematical expressions, such as Quantity * Price, in this clause.
- **FROM**—This is the second most important clause in the SELECT statement and is also required. You use the FROM clause to specify the tables or views from which to draw the columns you've listed in the SELECT clause. You can use this clause in more complex ways, but we'll discuss this in later chapters.

- **WHERE**—This is an optional clause that you use to filter the rows returned by the FROM clause. The WHERE keyword is followed by an expression, technically known as a *predicate*, that evaluates to true, false, or unknown. You can test the expression by using standard comparison operators, Boolean operators, or special operators. We'll discuss all the elements of the WHERE clause in Chapter 6.
- **GROUP BY**—When you use aggregate functions in the SELECT clause to produce summary information, you use the GROUP BY clause to divide the information into distinct groups. Your database system uses any column or list of columns following the GROUP BY keywords as grouping columns. The GROUP BY clause is optional, and we'll examine it further in Chapter 13, Grouping Data.
- **HAVING**—The HAVING clause filters the result of aggregate functions in grouped information. It is similar to the WHERE clause in that the HAVING keyword is followed by an expression that evaluates to true, false, or unknown. You can test the expression by using standard comparison operators, Boolean operators, or special operators. HAVING is also an optional clause, and we'll take a closer look at it in Chapter 14, Filtering Grouped Data.

We're going to work with a very basic SELECT statement at first, so we'll focus on the SELECT and FROM clauses. We'll add the other clauses, one by one, as we work through the other chapters to build more complex SELECT statements.

A Quick Aside: Data versus Information

Before we pose the first query to the database, one thing must be perfectly clear: There is a distinct difference between *data* and *information*. In essence, data is what you store in the database, and information is what you retrieve from the database. This distinction is important for you to understand because it helps you to keep things in proper perspective. Remember that a database is designed to provide meaningful information to someone within your organization. However, the information can be provided only if the appropriate data exists in the database and if the database itself has been structured in such a way to support that information. Let's examine these terms in more detail.

The values that you store in the database are data. Data is static in the sense that it remains in the same state until you modify it by some manual or automated process. Figure 4–2 shows some sample data.

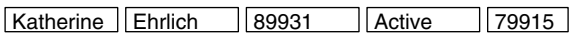


Figure 4–2 *An example of basic data*

On the surface, this data is meaningless. For example, there is no easy way for you to determine what 89931 represents. Is it a ZIP Code? Is it a part number? Even if you know it represents a customer identification number, is it associated with Katherine Ehrlich? There's no way to know until the data is processed. After you process the data so that it is meaningful and useful when you work with it or view it, the data becomes information. Information is dynamic in that it constantly changes relative to the data stored in the database and also in its ability to be processed and presented in an unlimited number of ways. You can show information as the result of a SELECT statement, display it in a form on your computer screen, or print it on paper as a report. But the point to remember is that you must process your data in a manner that enables you to turn it into meaningful information.

Figure 4–3 shows the data from the previous example transformed into information on a customer screen. This illustrates how the data can be manipulated in such a way that it is now meaningful to anyone who views it.

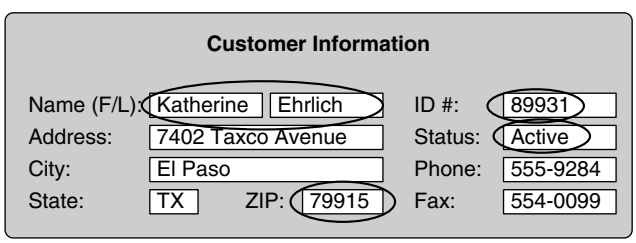


Figure 4–3 *An example of data processed into information*

When you work with a SELECT statement, you use its clauses to manipulate *data*, but the statement itself returns *information*. Get the picture?

There's one last issue we need to address. When you execute a SELECT statement, it usually retrieves one or more rows of information—the exact

number depends on how you construct the statement. These rows are collectively known as a *result set*, which is the term we use throughout the remainder of the book. This name makes perfect sense because you always work with sets of data whenever you use a relational database. (Remember that the relational model is based, in part, on set theory.) You can easily view the information in a result set and, in many cases, you can modify its data. But, once again, it all depends on how you construct your SELECT statement.

So let's get down to business and start using the SELECT statement.

Translating Your Request into SQL

When you request information from the database, it's usually in the form of a question or a statement that implies a question. For example, you might formulate statements such as these:

"Which cities do our customers live in?"

"Show me a current list of our employees and their phone numbers."

"What kind of classes do we currently offer?"

"Give me the names of the folks on our staff and the dates they were hired."

After you know what you want to ask, you can translate your request into a more formal statement. You compose the translation using this form:

Select <item> from the <source>

Start by looking at your request and replacing words or phrases such as *"list*," *"show me*," *"what*," *"which*," and *"who"* with the word "*Select*." Next, identify any nouns in your request, and determine whether a given noun represents an item you want to see or the name of a table in which an item might be stored. If it's an item, use it as a replacement for <item> in the translation statement. If it's a table name, use it as a replacement for <source>. If you translate the first question listed earlier, your statement looks something like this:

Select city from the customers table

After you define your translation statement, you need to turn it into a full-fledged SELECT statement using the SQL syntax shown in Figure 4–4. The

first step, however, is to clean up your translation statement. You do so by crossing out any word that is not a noun representing the name of a column or table or that is not a word specifically used in the SQL syntax. Here's how the translation statement looks during the process of cleaning it up:

Select city from ~~the~~ customers ~~table~~

Remove the words you've crossed out, and you now have a complete SELECT statement.

```
SELECT City FROM Customers
```

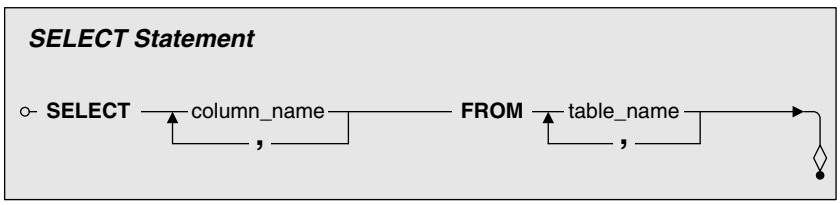


Figure 4–4 *The syntax of a simple SELECT statement*

You can use the three-step technique we just presented on any request you send to your database. In fact, we use this technique throughout most of the book, and we encourage you to use it while you're beginning to learn how to build these statements. However, you'll eventually merge these steps into one seamless operation as you get more accustomed to writing SELECT statements.

Remember that you'll work mostly with columns and tables when you're beginning to learn how to use SQL. The syntax diagram in Figure 4–4 reflects this fact by using column_name in the SELECT clause and table_name in the FROM clause. In the next chapter, you'll learn how to use other terms in these clauses to create more complex SELECT statements.

You probably noticed that the request we used in the previous example is relatively straightforward. It was easy to both redefine it as a translation statement and identify the column names that were present in the statement. But what if a request is not as straightforward and easy to translate, and it's difficult to identify the columns you need for the SELECT clause? The easiest course of action is to refine your request and make it more specific. For example, you can refine a request such as *"Show me the information on our clients"* by recasting it more clearly as *"List the name, city, and phone number for each*

of our clients." If refining the request doesn't solve the problem, you still have two other options. Your first alternative is to determine whether the table specified in the FROM clause of the SELECT statement contains any column names that can help to clarify the request and thus make it easier to define a translation statement. Your second alternative is to examine the request more closely and determine whether a word or phrase it contains *implies* any column names. Whether you can use either or both alternatives depends on the request itself. Just remember that you do have techniques available when you find it difficult to define a translation statement. Let's look at an example of each technique and how you can apply it in a typical scenario.

To illustrate the first technique, let's say you're trying to translate the following request.

> *"I need the names and addresses of all our employees."*

This looks like a straightforward request on the surface. But if you review this request again, you'll find one minor problem: Although you can determine the table you need (Employees) for the translation statement, there's nothing within the request that helps you identify the specific columns you need for the SELECT clause. Although the words *"names"* and *"addresses"* appear in the request, they are terms that are general in nature. You can solve this problem by reviewing the table you identified in the request and determining whether it contains any columns you can substitute for these terms. If so, use the column names in the translation statement. (You can opt to use generic versions of the column names in the translation statement if it will help you visualize the statement more clearly. However, you will need to use the actual column names in the SQL syntax.) In this case, look for column names in the Employees table shown in Figure 4–5 that could be used in place of the words "names" and "addresses."

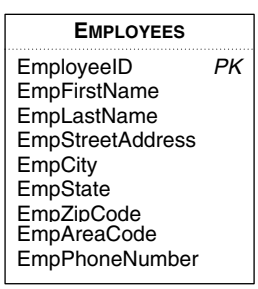


Figure 4–5 *The structure of the Employees table*

To fully satisfy the need for "names" and "addresses," you will indeed use six columns from this table. EmpFirstName and EmpLastName will both replace "names" in the request, and EmpStreetAddress, EmpCity, EmpState, and EmpZipCode will replace "addresses." Now, apply the entire translation process to the request, which we've repeated for your convenience. (We'll use generic forms of the column names for the translation statement and the actual column names in the SQL syntax.)

"I need the names and addresses of all our employees."

Translation	Select first name, last name, street address, city, state, and ZIP Code from the employees table
Clean Up	Select first name, last name, street address, city, state, ~~and~~ ZIP Code from ~~the~~ employees ~~table~~
SQL	`SELECT EmpFirstName, EmpLastName, EmpStreetAddress, EmpCity, EmpState, EmpZipCode FROM Employees`

❖ **Note** This example clearly illustrates how to use multiple columns in a SELECT clause. We'll discuss this technique in more detail later in this section.

The next example illustrates the second technique, which involves searching for implied columns within the request. Let's assume you're trying to put the following request through the translation process.

"What kind of classes do we currently offer?"

At first glance, it might seem difficult to define a translation statement from this request. The request doesn't indicate any column names, and without even one item to select, you can't create a complete translation statement. What do you do now? Take a closer look at each word in the request and determine whether there is one that *implies* a column name within the Classes table. Before you read any further, take a moment to study the request again. Can you find such a word?

In this case, the word "kind" might imply a column name in the Classes table. Why? Because a kind of class can also be thought of as a category of class. If there is a category column in the Classes table, then you have the column

name you need to complete the translation statement and, by inference, the SELECT statement. Let's assume that there is a category column in the Classes table and take the request through the three-step process once again.

"What kind of classes do we currently offer?"

Translation	Select category from the classes table
Clean Up	Select category from ~~the~~ classes ~~table~~
SQL	`SELECT Category` `FROM Classes`

As the example shows, this technique involves using synonyms as replacements for certain words or phrases within the request. If you identify a word or phrase that might imply a column name, try to replace it with a synonym. The synonym you choose might indeed identify a column that exists in the database. However, if the first synonym that comes to mind doesn't work, try another. Continue this process until you either find a synonym that does identify a column name or until you're satisfied that neither the original word nor any of its synonyms represent a column name.

❖ **Note** Unless we indicate otherwise, all column names and table names used in the SQL syntax portion of the examples are drawn from the sample databases in Appendix B, Schema for the Sample Databases. This convention applies to all examples for the remainder of the book.

Expanding the Field of Vision

You can retrieve multiple columns within a SELECT statement as easily as you can retrieve a single column. List the names of the columns you want to use in the SELECT clause, and separate each name in the list with a comma. In the syntax diagram shown in Figure 4–6, the option to use more than one column is indicated by a line that flows from right to left beneath column_name. The comma in the middle of the line denotes that you must insert a comma before the next column name you want to use in the SELECT clause.

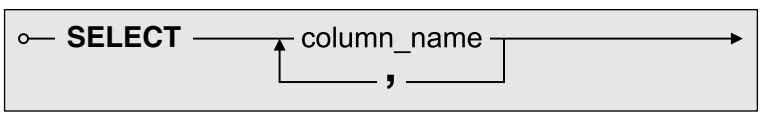


Figure 4–6 *The syntax for using multiple columns in a SELECT clause*

The option to use multiple columns in the SELECT statement provides you with the means to answer questions such as these.

"Show me a current list of our employees and their phone numbers."

Translation	Select the last name, first name, and phone number of all our employees from the employees table
Clean Up	Select ~~the~~ last name, first name, ~~and~~ phone number ~~of all our employees~~ from ~~the~~ employees ~~table~~
SQL	`SELECT EmpLastName, EmpFirstName, EmpPhoneNumber FROM Employees`

"What are the names and prices of the products we carry, and under what category is each item listed?"

Translation	Select the name, price, and category of every product from the products table
Clean Up	Select ~~the~~ name, price, ~~and~~ category ~~of every product~~ from ~~the~~ products ~~table~~
SQL	`SELECT ProductName, RetailPrice, Category FROM Products`

You gain the advantage of seeing a wider spectrum of information when you work with several columns in a SELECT statement. Incidentally, the sequence of the columns in your SELECT clause is not important—you can list the columns in any order you want. This gives you the flexibility to view the same information in a variety of ways.

For example, let's say you're working with the table shown in Figure 4–7, and you're asked to pose the following request to the database.

"Show me a list of subjects, the category each belongs to, and the code we use in our catalog. But I'd like to see the name first, followed by the category and then the code."

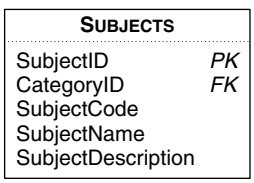


Figure 4–7 *The structure of the Subjects table*

You can still transform this request into an appropriate SELECT statement, even though the person making the request wants to see the columns in a specific order. Just list the column names in the order specified when you define the translation statement. Here's how the process looks when you transform this request into a SELECT statement.

Translation	Select the subject name, category ID, and subject code from the subjects table
Clean Up	Select ~~the~~ subject name, category ID, ~~and~~ subject code from ~~the~~ subjects ~~table~~
SQL	`SELECT SubjectName, CategoryID, SubjectCode` `FROM Subjects`

Using a Shortcut to Request All Columns

There is no limit to the number of columns you can specify in the SELECT clause—in fact, you can list all the columns from the source table. The following example shows the SELECT statement you use to specify all the columns from the Subjects table in Figure 4–7.

```
SQL         SELECT SubjectID, CategoryID, SubjectCode,
               SubjectName, SubjectDescription
            FROM Subjects
```

When you specify all the columns from the source table, you'll have a lot of typing to do if the table contains a number of columns! Fortunately, the SQL Standard specifies the asterisk as a shortcut you can use to shorten the statement considerably. The syntax diagram in Figure 4–8 shows that you can use the asterisk as an alternative to a list of columns in the SELECT clause.

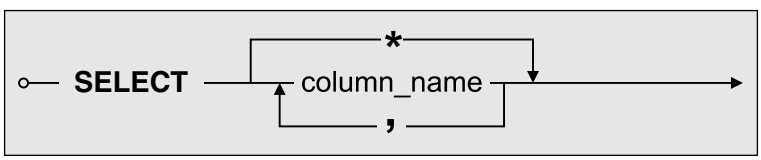


Figure 4–8 *The syntax for the asterisk shortcut*

Place the asterisk immediately after the SELECT clause when you want to specify all the columns from the source table in the FROM clause. For

example, here's how the preceding SELECT statement looks when you use the shortcut.

```
SQL         SELECT *
            FROM Subjects
```

You'll certainly do less typing with this statement! However, one issue arises when you create SELECT statements in this manner: The asterisk represents all of the columns that *currently exist* in the source table, and adding or deleting columns affects what you see in the result set of the SELECT statement. (Oddly enough, the SQL Standard states that adding or deleting columns *should not* affect your result set.) This issue is important only if you must see the same columns in the result set consistently. Your database system will not warn you if columns have been deleted from the source table when you use the asterisk in the SELECT clause, but it will raise a warning when it can't find a column you *explicitly* specified. Although this does not pose a real problem for our purposes, it will be an important issue when you delve into the world of programming with SQL. Our rule of thumb is this: Use the asterisk only when you need to create a "quick and dirty" query to see all the information in a given table. Otherwise, specify all the columns you need for the query. In the end, the query will return exactly the information you need and will be more self-documenting.

The examples we've seen so far are based on simple requests that require columns from only one table. You'll learn how to work with more complex requests that require columns from several tables in Part III.

Eliminating Duplicate Rows

When working with SELECT statements, you'll inevitably come across result sets with duplicate rows. There is no cause for alarm if you see such a result set. Use the DISTINCT keyword in your SELECT statement, and the result set will be free and clear of all duplicate rows. Figure 4–9 shows the syntax diagram for the DISTINCT keyword.

As the diagram illustrates, DISTINCT is an optional keyword that precedes the list of columns specified in the SELECT clause. The DISTINCT keyword asks your database system to evaluate the values of all the columns *as a single unit* on a row-by-row basis and eliminate any redundant rows it finds. The remaining unique rows are then returned to the result set. The following

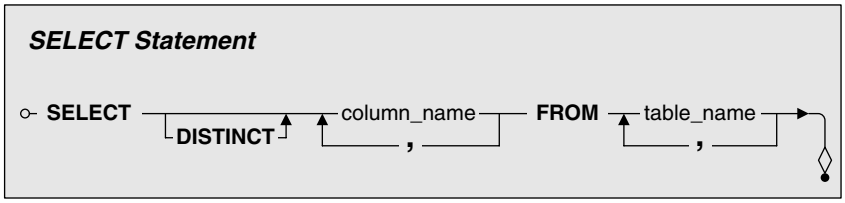


Figure 4–9 *The syntax for the DISTINCT keyword*

example shows what a difference the DISTINCT keyword can make under the appropriate circumstances.

Let's say you're posing the following request to the database.

"Which cities are represented by our bowling league membership?"

The question seems easy enough, so you take it through the translation process.

Translation	Select city from the bowlers table
Clean Up	Select city from ~~the~~ bowlers ~~table~~
SQL	`SELECT City` `FROM Bowlers`

The problem is that the result set for this SELECT statement shows *every occurrence* of each city name found in the Bowlers table. For example, if there are 20 people from Bellevue and 7 people from Kent and 14 people from Seattle, the result set displays 20 occurrences of Bellevue, 7 occurrences of Kent, and 14 occurrences of Seattle. Clearly, this redundant information is unnecessary. All you want to see is a *single* occurrence of each city name found in the Bowlers table. You resolve this problem by using the DISTINCT keyword in the SELECT statement to eliminate the redundant information.

Let's run the request through the translation process once again using the DISTINCT keyword. Note that we now include the word "distinct" in both the Translation step and the Clean Up step.

"Which cities are represented by our bowling league membership?"

Translation	Select the distinct city values from the bowlers table
Clean Up	Select ~~the~~ distinct city ~~values~~ from ~~the~~ bowlers ~~table~~
SQL	`SELECT DISTINCT City` `FROM Bowlers`

The result set for this SELECT statement displays exactly what you're looking for—a single occurrence of each distinct (or unique) city found in the Bowlers table.

You can use the DISTINCT keyword on multiple columns as well. Let's modify the previous example by requesting both the city and the state from the Bowlers table. Our new SELECT statement looks like this.

```
SELECT DISTINCT City, State FROM Bowlers
```

This SELECT statement returns a result set that contains unique records and shows definite distinctions between cities with the same name. For example, it shows the distinction between "Portland, ME," "Portland, OR," "Hollywood, CA," and "Hollywood, FL." It's worthwhile to note that most database systems sort the output in the sequence in which you specify the columns, so you'll see these values in the sequence "Hollywood, CA," "Hollywood, FL," "Portland, ME," and "Portland, OR." However, the SQL Standard does not require the result to be sorted in this order. If you want to guarantee the sort sequence, read on to the next section to learn about the ORDER BY clause.

The DISTINCT keyword is a very useful tool under the right circumstances. Use it only when you really want to see unique rows in your result set.

❖ **Caution** For database systems that include a graphical interface, you can usually request that the result set of a query be displayed in an updatable grid of rows and columns. You can type a new value in a column on a row, and the database system updates the value stored in your table. (Your database system actually executes an UPDATE query on your behalf behind the scenes—you can read more about that in Chapter 15, Updating Sets of Data.)

However, in all database systems that we studied, when you include the DISTINCT keyword, the resulting set of rows cannot be updated. To be able to update a column in a row, your database system needs to be able to uniquely identify the specific row and column you want to change. When you use DISTINCT, the values you see in each row are the result of evaluating perhaps dozens of duplicate rows. If you try to update one of the columns, your database won't know which specific row to change. Your database system also doesn't know if perhaps you mean to change all the rows with the same duplicate value.

Sorting Information

At the beginning of this chapter, we said that the SELECT operation can be broken down into three smaller operations: the SELECT statement, the SELECT expression, and the SELECT query. We also stated that you can combine these operations in various ways to answer complex requests. However, you also need to combine these operations in order to sort the rows of a result set.

By definition, the rows of a result set returned by a SELECT statement are unordered. The sequence in which they appear is typically based on their physical position in the table. (The actual sequence is often determined dynamically by your database system based on how it decides to most efficiently satisfy your request.) The only way to sort the result set is to embed the SELECT statement within a SELECT query, as shown in Figure 4–10. We define a SELECT query as a SELECT statement with an ORDER BY clause. The ORDER BY clause of the SELECT query lets you specify the sequence of rows in the final result set. As you'll learn in later chapters, you can actually embed a SELECT statement within another SELECT statement or SELECT expression to answer very complex questions. However, the SELECT query cannot be embedded at any level.

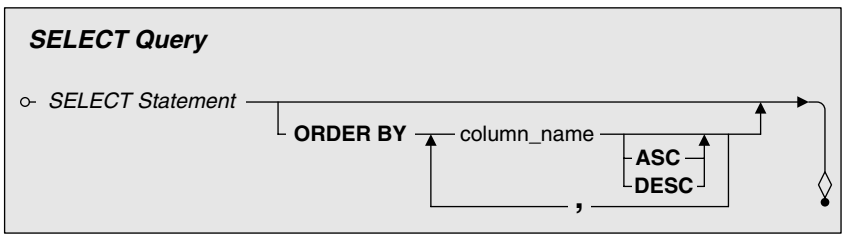


Figure 4–10 *The syntax diagram for the SELECT query*

❖ **Note** Throughout this book, we use the same terms you'll find in the SQL Standard or in common usage in most database systems. The 2003 SQL Standard, however, defines the ORDER BY clause as part of a *cursor* (an object that you define inside an application program), as part of an *array* (a list of values that form a logical table such as a *subquery,* discussed in Chapter 11, Subqueries), or as part of a *scalar subquery* (a subquery that returns only one value). A complete discussion of cursors and arrays is beyond the scope of this book. Because nearly all implementations of SQL allow you to include an ORDER BY clause at the end of a SELECT statement that you can save in a view, we invented the term *SELECT query* to describe this type of statement. This also allows us to discuss the concept of sorting the final output of a query for display online or for use in a report. It's our understanding that the draft 2007/2008 standard does allow using ORDER BY in more places, but we'll use this separate construct in this book to cover the topic.

The ORDER BY clause allows you to sort the result set of the specified SELECT statement by one or more columns and also provides the option of specifying an ascending or descending sort order for each column. The only columns you can use in the ORDER BY clause are those that are currently listed in the SELECT clause. (Although this requirement is specified in the SQL Standard, some vendor implementations allow you to disregard it completely. However, we comply with this requirement in all the examples used throughout the book.) When you use two or more columns in an ORDER BY clause, separate each column with a comma. The SELECT query returns a final result set once the sort is complete.

❖ **Note** The ORDER BY clause *does not* affect the physical order of the rows in a table. If you do need to change the physical order of the rows, refer to your database software's documentation for the proper procedure.

First Things First: Collating Sequences

Before we look at some examples using the SELECT query, a brief word on collating sequences is in order. The manner in which the ORDER BY clause sorts the information depends on the collating sequence used by your database software. The collating sequence determines the order of precedence for every character listed in the current language character set specified by

your operating system. For example, it identifies whether lowercase letters will be sorted before uppercase letters, or whether case will even matter. Check your database software's documentation, and perhaps consult your database administrator to determine the default collating sequence for your database. For more information on collating sequences, see the subsection Comparing String Values: A Caution in Chapter 6.

Let's Now Come to Order

With the availability of the ORDER BY clause, you can present the information you retrieve from the database in a more meaningful fashion. This applies to simple requests as well as complex ones. You can now rephrase your requests so that they also indicate sorting requirements. For example, a question such as *"What are the categories of classes we currently offer?"* can be restated as *"List the categories of classes we offer and show them in alphabetical order."*

Before beginning to work with the SELECT query, you need to adjust the way you define a translation statement. This involves adding a new section at the end of the translation statement to account for the new sorting requirements specified within the request. Use this new form to define the translation statement.

Select <item> from the <source> **and order by <column(s)>**

Now that your request will include phrases such as "sort the results by city," "show them in order by year," or "list them by last name and first name," study the request closely to determine which column or columns you need to use for sorting purposes. This is a simple exercise because most people use these types of phrases, and the columns needed for the sort are usually self-evident. After you identify the appropriate column or columns, use them as a replacement for <column(s)> in the translation statement. Let's take a look at a simple request to see how this works.

"List the categories of classes we offer and show them in alphabetical order."

Translation	Select category from the classes table and order by category
Clean Up	Select category from ~~the~~ classes ~~table and~~ order by category

SQL

```
SELECT Category
FROM Classes
ORDER BY Category
```

In this example, you can assume that Category will be used for the sort because it's the only column indicated in the request. You can also assume that the sort should be in ascending order because there's nothing in the request to indicate the contrary. This is a safe assumption. According to the SQL Standard, ascending order is automatically assumed if you don't specify a sort order. However, if you want to be absolutely explicit, insert ASC after Category in the ORDER BY clause.

In the following request, the column needed for the sort is more clearly defined.

"Show me a list of vendor names in ZIP Code order."

Translation Select vendor name and ZIP Code
from the vendors table and order by ZIP Code

Clean Up Select vendor name ~~and~~ ZIP Code
from ~~the~~ vendors ~~table and~~ order by ZIP Code

```
SQL         SELECT VendName, VendZipCode
            FROM Vendors
            ORDER BY VendZipCode
```

In general, most people will tell you if they want to see their information in descending order. When this situation arises and you need to display the result set in reverse order, insert the DESC keyword after the appropriate column in the ORDER BY clause. For example, here's how you would modify the SELECT statement in the previous example when you want to see the information sorted by ZIP Code in descending order.

```
SQL         SELECT VendName, VendZipCode
            FROM Vendors
            ORDER BY VendZipCode DESC
```

The next example illustrates a more complex request that requires a multicolumn sort. The only difference between this example and the previous two examples is that this example uses more columns in the ORDER BY clause. Note that the columns are separated with commas, which is in accordance with the syntax diagram shown in Figure 4–10.

"Display the names of our employees, including their phone number and ID number, and list them by last name and first name."

Translation: Select last name, first name, phone number, and employee ID from the employees table and order by last name and first name

Clean Up: Select last name, first name, phone number, ~~and~~ employee ID from ~~the~~ employees ~~table and~~ order by last name ~~and~~ first name

SQL:
```
SELECT EmpLastName, EmpFirstName,
   EmpPhoneNumber, EmployeeID
FROM Employees
ORDER BY EmpLastName, EmpFirstName
```

One of the interesting things you can do with the columns in an ORDER BY clause is to specify a different sort order for each column. In the previous example, you can specify a descending sort for the column containing the last name and an ascending sort for the column containing the first name. Here's how the SELECT statement looks when you make the appropriate modifications.

SQL:
```
SELECT EmpLastName, EmpFirstName, EmpPhoneNumber,
   EmployeeID
FROM Employees
ORDER BY EmpLastName DESC, EmpFirstName ASC
```

Although you don't need to use the ASC keyword explicitly, the statement is more self-documenting if you include it.

The previous example brings an interesting question to mind: Is any importance placed on the sequence of the columns in the ORDER BY clause? The answer is *"Yes!"* The sequence is important because your database system will evaluate the columns in the ORDER BY clause from left to right. Also, the importance of the sequence grows in direct proportion to the number of columns you use. Always sequence the columns in the ORDER BY clause properly so that the result sorts in the appropriate order.

❖ **Note** The database products from Microsoft (Microsoft Office Access and Microsoft SQL Server) include an interesting extension that allows you to request a subset of rows based on your ORDER BY clause by using the TOP keyword in the SELECT clause. For example, you can find out the five most expensive products in the Sales Orders database by requesting:

```
SELECT TOP 5 ProductName, RetailPrice
   FROM Products
   ORDER BY RetailPrice DESC
```

The database sorts all the rows from the Products table descending by price and then returns the top five rows. Both database systems also allow you to specify the number of rows returned as a percentage of all the rows. For example, you can find out the top 10 percent of products by price by requesting:

```
SELECT TOP 10 PERCENT ProductName, RetailPrice
   FROM Products
   ORDER BY RetailPrice DESC
```

In fact, if you want to specify ORDER BY in a view, SQL Server requires that you include the TOP keyword. If you want all rows, you must specify TOP 100 PERCENT. For this reason, you'll see that all the sample views in SQL Server that include an ORDER BY clause also specify TOP 100 PERCENT. There is no such restriction in Microsoft Access.

Saving Your Work

Save your SELECT statements—every major database software program provides a way for you to save them! Saving your statements eliminates the need to recreate them every time you want to make the same request to the database. When you save your SELECT statement, assign a meaningful name that will help you remember what type of information the statement provides. And if your database software allows you to do so, write a concise description of the statement's purpose. The value of the description will become quite clear when you haven't seen a particular SELECT statement for some time and you need to remember why you constructed it in the first place.

A saved SELECT statement is categorized as a query in some database programs and as a view, function, or stored procedure in others. Regardless of its

designation, every database program provides you with a means to execute, or run, the saved statement and work with its result set.

❖ **Note** For the remainder of this discussion, we'll use the word *query* to represent the saved SELECT statement and *execute* to represent the method used to work with it.

Two common methods are used to execute a query. The first is an interactive device (such as a command on a toolbar or query grid), and the second is a block of programming code. You'll use the first method quite extensively. There's no need to worry about the second method until you begin working with your database software's programming language. Although it's our job to teach you how to create and use SQL statements, it's your job to learn how to create, save, and execute them in your database software program.

Sample Statements

Now that we've covered the basic characteristics of the SELECT statement and SELECT query, let's take a look at some examples of how these operations are applied in different scenarios. These examples encompass each of the sample databases, and they illustrate the use of the SELECT statement, the SELECT query, and the two supplemental techniques used to establish columns for the translation statement. We've also included sample result sets that would be returned by these operations and placed them immediately after the SQL syntax line. The name that appears immediately above a result set has a twofold purpose: It identifies the result set itself, and it is also the name that we assigned to the SQL statement in the example.

In case you're wondering why we assigned a name to each SQL statement, it's because we saved them! In fact, we've named and saved all the SQL statements that appear in the examples here and throughout the remainder of the book. Each is stored in the appropriate sample database (as indicated within the example), and we prefixed the names of the queries relevant to this chapter with "CH04." You can follow the instructions in the Introduction of this book to load the samples onto your computer. This gives you the opportunity to see these statements in action before you try your hand at writing them yourself.

❖ **Note** Just a reminder: All the column names and table names used in these examples are drawn from the sample database structures shown in Appendix B.

Sales Orders Database

"Show me the names of all our vendors."

Translation Select the vendor name from the vendors table

Clean Up Select ~~the~~ vendor name from ~~the~~ vendors ~~table~~

SQL
```
SELECT VendName
FROM Vendors
```

CH04_Vendor_Names (10 Rows)

VendName
Shinoman, Incorporated
Viscount
Nikoma of America
ProFormance
Kona, Incorporated
Big Sky Mountain Bikes
Dog Ear
Sun Sports Suppliers
Lone Star Bike Supply
Armadillo Brand

"What are the names and prices of all the products we carry?"

Translation Select product name, retail price from the products table

Clean Up Select product name, retail price from ~~the~~ products ~~table~~

SQL
```
SELECT ProductName, RetailPrice
FROM Products
```

CH04_Product_Price_List (40 Rows)

ProductName	Retail Price
Trek 9000 Mountain Bike	$1,200.00
Eagle FS-3 Mountain Bike	$1,800.00
Dog Ear Cyclecomputer	$75.00
Victoria Pro All Weather Tires	$54.95
Dog Ear Helmet Mount Mirrors	$7.45
Viscount Mountain Bike	$635.00
Viscount C-500 Wireless Bike Computer	$49.00
Kryptonite Advanced 2000 U-Lock	$50.00
Nikoma Lok-Tight U-Lock	$33.00
Viscount Microshell Helmet	$36.00
<< more rows here >>	

"Which states do our customers come from?"

Translation Select the distinct state values from the customers table

Clean Up Select ~~the~~ distinct state ~~values~~ from ~~the~~ customers ~~table~~

SQL

```
SELECT DISTINCT CustState
FROM Customers
```

CH04_Customer_States (4 Rows)

CustState
CA
OR
TX
WA

Entertainment Agency Database

"List all entertainers and the cities they're based in, and sort the results by city and name in ascending order."

Translation: Select city and stage name from the entertainers table and order by city and stage name

Clean Up: Select city ~~and~~ stage name from ~~the~~ entertainers ~~table and~~ order by city ~~and~~ stage name

SQL:

```
SELECT EntCity, EntStageName
FROM Entertainers
ORDER BY EntCity ASC, EntStageName ASC
```

CH04_Entertainer_Locations (13 Rows)

EntCity	EntStageName
Auburn	Caroline Coie Cuartet
Auburn	Topazz
Bellevue	Jazz Persuasion
Bellevue	Jim Glynn
Bellevue	Susan McLain
Redmond	Carol Peacock Trio
Redmond	JV & the Deep Six
Seattle	Coldwater Cattle Company
Seattle	Country Feeling
Seattle	Julia Schnebly
<< more rows here >>	

"Give me a unique list of engagement dates. I'm not concerned with how many engagements there are per date."

Translation Select the distinct start date values from the engagements table

Clean Up Select ~~the~~ distinct start date ~~values~~ from ~~the~~ engagements ~~table~~

SQL
```
SELECT DISTINCT StartDate
FROM Engagements
```

CH04_Engagement_Dates (64 Rows)

StartDate
2007-09-01
2007-09-10
2007-09-11
2007-09-15
2007-09-17
2007-09-18
2007-09-24
2007-09-29
2007-09-30
2007-10-01
<< more rows here >>

School Scheduling Database

"*Can we view complete class information?*"

Translation Select all columns from the classes table

Clean Up Select ~~all columns~~ * from ~~the~~ classes ~~table~~

SQL
```
SELECT *
FROM Classes
```

CH04_Class_Information (76 Rows)

ClassID	SubjectID	ClassRoomID	Credits	StartTime	Duration	<<other columns>>
1000	11	1231	5	10:00	50	...
1002	12	1619	4	15:30	110	...
1004	13	1627	4	08:00	50	...
1006	13	1627	4	09:00	110	...
1012	14	1627	4	13:00	170	...
1020	15	3404	4	13:00	110	...
1030	16	1231	5	11:00	50	...
1031	16	1231	5	14:00	50	...
1156	37	3443	5	08:00	50	...
1162	37	3443	5	09:00	80	...
<< *more rows here* >>						

"Give me a list of the buildings on campus and the number of floors for each building. Sort the list by building in ascending order."

Translation Select building name and number of floors from the buildings table, ordered by building name

Clean Up Select building name ~~and~~ number of floors from ~~the~~ buildings ~~table,~~ order~~ed~~ by building name

SQL
```
SELECT BuildingName, NumberOfFloors
FROM Buildings
ORDER BY BuildingName ASC
```

CH04_Building_List (6 Rows)

BuildingName	NumberOfFloors
Arts and Sciences	3
College Center	3
Instructional Building	3
Library	2
PE and Wellness	1
Technology Building	2

Bowling League Database

"Where are we holding our tournaments?"

Translation Select the distinct tourney location values from the tournaments table

Clean Up Select ~~the~~ distinct tourney location ~~values~~ from ~~the~~ tournaments ~~table~~

SQL

```
SELECT DISTINCT TourneyLocation
FROM Tournaments
```

CH04_Tourney_Locations (7 Rows)

TourneyLocation
Acapulco Lanes
Bolero Lanes
Imperial Lanes
Red Rooster Lanes
Sports World Lanes
Thunderbird Lanes
Totem Lanes

"Give me a list of all tournament dates and locations. I need the dates in descending order and the locations in alphabetical order."

Translation Select tourney date and location from the tournaments table and order by tourney date in descending order and location in ascending order

Clean Up Select tourney date ~~and~~ location from ~~the~~ tournaments ~~table and~~ order by tourney date ~~in~~ descending ~~order and~~ location ~~in~~ ascending ~~order~~

SQL
```
SELECT TourneyDate, TourneyLocation
FROM Tournaments
ORDER BY TourneyDate DESC, TourneyLocation ASC
```

CH04_Tourney_Dates (14 Rows)

TourneyDate	TourneyLocation
2008-08-15	Totem Lanes
2008-08-08	Imperial Lanes
2008-08-01	Sports World Lanes
2008-07-25	Bolero Lanes
2008-07-18	Thunderbird Lanes
2008-07-11	Red Rooster Lanes
2007-12-04	Acapulco Lanes
2007-11-27	Totem Lanes
2007-11-20	Sports World Lanes
2007-11-13	Imperial Lanes
<< more rows here >>	

Recipes Database

"What types of recipes do we have, and what are the names of the recipes we have for each type? Can you sort the information by type and recipe name?"

Translation Select recipe class ID and recipe title from the recipes table and order by recipe class ID and recipe title

Clean Up Select recipe class ID ~~and~~ recipe title from ~~the~~ recipes ~~table and~~ order by recipeclass ID ~~and~~ recipe title

SQL

```
SELECT RecipeClassID, RecipeTitle
FROM Recipes
ORDER BY RecipeClassID ASC, RecipeTitle ASC
```

CH04_Recipe_Classes_And_Titles (15 Rows)

RecipeClassID	RecipeTitle
1	Fettuccini Alfredo
1	Huachinango Veracruzana (Red Snapper, Veracruz style)
1	Irish Stew
1	Pollo Picoso
1	Roast Beef
1	Salmon Filets in Parchment Paper
1	Tourtière (French-Canadian Pork Pie)
2	Asparagus
2	Garlic Green Beans
3	Yorkshire Pudding
<< more rows here >>	

"Show me a list of unique recipe class IDs in the recipes table."

Translation	Select the distinct recipe class ID values from the recipes table
Clean Up	Select ~~the~~ distinct recipe class ID ~~values~~ from ~~the~~ recipes ~~table~~
SQL	`SELECT DISTINCT RecipeClassID` `FROM Recipes`

CH04_Recipe_Class_Ids (6 Rows)

RecipeClassID
1
2
3
4
5
6

SUMMARY

In this chapter, we introduced the SELECT operation, and you learned that it is one of four data manipulation operations in SQL. (The others are UPDATE, INSERT, and DELETE, covered in Part V.) We also discussed how the SELECT operation can be divided into three smaller operations: the SELECT statement, the SELECT expression, and the SELECT query.

The discussion then turned to the SELECT statement, where you were introduced to its component clauses. We covered the fact that the SELECT and FROM clauses are the fundamental clauses required to retrieve information from the database and that the remaining clauses—WHERE, GROUP BY, and HAVING—are used to conditionally process and filter the information returned by the SELECT clause.

We briefly diverged into a discussion of the difference between data and information. You learned that the values stored in the database are data and that information is data that has been processed in a manner that makes it meaningful to the person viewing it. You also learned that the rows of information returned by a SELECT statement are known as a result set.

Retrieving information was the next topic of discussion, and we began by presenting the basic form of the SELECT statement. You learned how to build a proper SELECT statement by using a three-step technique that involves taking a request and translating it into proper SQL syntax. You also learned that you could use two or more columns in the SELECT clause to expand the scope of information you retrieve from your database. We followed this section with a quick look at the DISTINCT keyword, which you learned is the means for eliminating duplicate rows from a result set.

Next, we looked at the SELECT query and how it can be combined with a SELECT statement to sort the SELECT statement's result set. You learned that this is necessary because the SELECT query is the only SELECT operation that contains an ORDER BY clause. We went on to show that the ORDER BY clause is used to sort the information by one or more columns and that each column can have its own ascending or descending sort specification. A brief discussion on saving your SELECT statements followed, and you learned that you can save your statement as a query or a view for future use.

Finally, we presented a number of examples using various tables in the sample databases. The examples illustrated how the various concepts and techniques presented in this chapter are used in typical scenarios and applications. In the next chapter, we'll take a closer look at the SELECT clause and show you how to retrieve something besides information from a list of columns.

The following section presents a number of requests that you can work out on your own.

Problems for You to Solve

Below, we show you the request statement and the name of the solution query in the sample databases. If you want some practice, you can work out the SQL you need for each request and then check your answer with the query we saved in the samples. Don't worry if your syntax doesn't exactly match the syntax of the queries we saved—as long as your result set is the same.

Sales Orders Database

1. *"Show me all the information on our employees."*
   You can find the solution in CH04_Employee_Information (8 rows).

2. *"Show me a list of cities, in alphabetical order, where our vendors are located, and include the names of the vendors we work with in each city."*
   You can find the solution in CH04_Vendor_Locations (10 rows).

Entertainment Agency Database

1. *"Give me the names and phone numbers of all our agents, and list them in last name/first name order."*
   You can find the solution in CH04_Agent_Phone_List (9 rows).
2. *"Give me the information on all our engagements."*
   You can find the solution in CH04_Engagement_Information (111 rows).
3. *"List all engagements and their associated start dates. Sort the records by date in descending order and by engagement in ascending order."*
   You can find the solution in CH04_Scheduled_Engagements (111 rows).

School Scheduling Database

1. *"Show me a complete list of all the subjects we offer."*
   You can find the solution in CH04_Subject_List (56 rows).
2. *"What kinds of titles are associated with our faculty?"*
   You can find the solution in CH04_Faculty_Titles (3 rows).
3. *"List the names and phone numbers of all our staff, and sort them by last name and first name."*
   You can find the solution in CH04_Staff_Phone_List (27 rows).

Bowling League Database

1. *"List all of the teams in alphabetical order."*
   You can find the solution in CH04_Team_List (8 rows).
2. *"Show me all the bowling score information for each of our members."*
   You can find the solution in CH04_Bowling_Score_Information (1,344 rows).
3. *"Show me a list of bowlers and their addresses, and sort it in alphabetical order."*
   You can find the solution in CH04_Bowler_Names_Addresses (32 rows).

Recipes Database

1. *"Show me a list of all the ingredients we currently keep track of."*
   You can find the solution in CH04_Complete_Ingredients_List (79 rows).
2. *"Show me all the main recipe information, and sort it by the name of the recipe in alphabetical order."*
   You can find the solution in CH04_Main_Recipe_Information (15 rows).

5

Getting More Than Simple Columns

"Facts are stubborn things."
—Tobias Smollett
Gil Blas de Santillane

Topics Covered in This Chapter

What Is an Expression?

What Type of Data Are You Trying to Express?

Changing Data Types: The CAST Function

Specifying Explicit Values

Types of Expressions

Using Expressions in a SELECT Clause

That "Nothing" Value: Null

Sample Statements

Summary

Problems for You to Solve

In Chapter 4, Creating a Simple Query, you learned how to use a SELECT statement to retrieve information from one or more columns in a table. This technique is useful if you're posing only simple requests to the database for some basic facts. However, you'll need to expand your SQL vocabulary when you begin working with complex requests. In this chapter, we'll introduce you to the concept of an expression as a way to manipulate the data in your tables to calculate or generate new columns of information. Next, we'll discuss how the *type* of data stored in a column can have an important impact on your queries and the expressions you create. We'll take a brief detour to

the CAST function, which you can use to actually change the type of data you're including in your expressions. You'll learn to create a constant (or literal) value that you can use in creative ways in your queries. You'll learn to use literals and values from columns in your table to create expressions. You'll learn how to adjust the scope of information you retrieve with a SELECT statement by using expressions to manipulate the data from which the information is drawn. Finally, you'll explore the special Null value and learn how it can impact how you work with expressions that use columns from your tables.

What Is an Expression?

To get more than simple columns, you need to create an expression. An *expression* is some form of operation involving numbers, character strings, or dates and times. It can use values drawn from specific columns in a table, constant (literal) values, or a combination of both. We'll show you how to generate literal values later in this chapter. After your database completes the operation defined by the expression, the expression returns a value to the SQL statement for further processing. You can use expressions to broaden or narrow the scope of the information you retrieve from the database. Expressions are especially useful when you are asking "what if" questions. Here's a sample of the types of requests you can answer using expressions.

"What is the total amount for each line item?"

"Give me a mailing list of employees, last name first."

"Show me the start time, end time, and duration for each class."

"Show the difference between the handicap score and the raw score for each bowler."

"What is the estimated per-hour rate for each engagement?"

"What if we raised the prices of our products by 5 percent?"

As you'll learn as you work through this chapter, expressions are a very valuable technique to add to your knowledge of SQL. You can use expressions to "slice and dice" the plain-vanilla data in your columns to create more meaningful results in your queries. You'll also find that expressions are very useful when you move on to Chapter 6, Filtering Your Data, and beyond. You'll use expressions to filter your data or to link data from related tables.

What Type of Data Are You Trying to Express?

The type of data used in an expression impacts the value the expression returns, so let's first look at some of the data types the SQL Standard provides. Every column in the database has an assigned *data type* that determines the kind of values the column can store. The data type also determines the operations that can be performed on the column's values. You need to understand the basic data types before you can begin to create literal values or combine columns and literals in an expression that is meaningful and that returns a proper value.

The SQL Standard defines seven general categories of types of data—character, national character, binary, numeric, Boolean, datetime, and interval. In turn, each category contains one or more uniquely named data types. Here's a brief look at each of these categories and their data types. (In the following list, we've broken the numeric category into two subcategories: exact numeric and approximate numeric.)

CHARACTER
: The character data type stores a fixed- or varying-length character string of one or more printable characters. The characters it accepts are usually based upon the American Standard Code for Information Interchange (ASCII) or the Extended Binary Coded Decimal Interchange Code (EBCDIC) character sets. A fixed-length character data type is known as CHARACTER or CHAR, and a varying-length character data type is known as CHARACTER VARYING, CHAR VARYING, or VARCHAR. You can define the length of data that you want to store in a character data type, but the maximum length that you can specify is defined by your database system. (This rule applies to the national character data types as well.) When the length of a character string exceeds a system-defined maximum (usually 255 or 1,024 characters), you must use a CHARACTER LARGE OBJECT, CHAR LARGE OBJECT, or CLOB data type. In many systems, the alias for CLOB is TEXT or MEMO.

NATIONAL CHARACTER
: The national character data type is the same as the character data type except that it draws its characters from ISO-defined foreign language character sets. NATIONAL CHARACTER, NATIONAL CHAR, and

	NCHAR are names used to refer to a fixed-length national character, and NATIONAL CHARACTER VARYING, NATIONAL CHAR VARYING, and NCHAR VARYING are names used to refer to a varying-length national character. When the length of a character string exceeds a system-defined maximum (usually 255 or 1,024 characters), you must use a NATIONAL CHARACTER LARGE OBJECT, NCHAR LARGE OBJECT, or NCLOB data type. In many systems, the alias for NCLOB is NTEXT.
BINARY	Use the BINARY LARGE OBJECT (or BLOB) data type to store binary data such as images, sounds, videos, or complex embedded documents such as word processing files or spreadsheets. In many systems, the names used for this data type include BINARY, BIT, and BIT VARYING.
EXACT NUMERIC	This data type stores whole numbers and numbers with decimal places. The precision (the number of significant digits) and the scale (the number of digits to the right of the decimal place) of an exact numeric can be user-defined and can only be equal to or less than the maximum limits allowed by the database system. NUMERIC, DECIMAL, DEC, SMALLINT, INTEGER, INT, and BIGINT are all names used to refer to this data type. One point you must remember is that the SQL Standard—as well as most database systems—defines a BIGINT as having a greater range of values than INTEGER, and INTEGER as having a greater range of values than a SMALLINT. Check your database system's documentation for the applicable ranges. Some systems also support a TINYINT data type that can hold a smaller range of values than SMALLINT.
APPROXIMATE NUMERIC	This data type stores numbers with decimal places and exponential numbers. Names used to refer to this data type include FLOAT, REAL, and DOUBLE PRECISION. The approximate numeric data types don't have a precision and scale per se, but the SQL Standard does allow a user-defined precision only for a FLOAT data type. Any scale associated with these data types is always defined by the database system. Note that the SQL Standard and most database systems define

	the range of values for a DOUBLE PRECISION data type to be greater than those of a REAL or FLOAT data type. Check your documentation for these ranges as well.
BOOLEAN	This data type stores true and false values, usually in a single binary bit. Some systems use BIT, INT, or TINYINT to store this data type.
DATETIME	Dates, times, and combinations of both are stored in this data type. The SQL Standard defines the date format as year-month-day and specifies time values as being based on a 24-hour clock. Although most database systems allow you to use the more common month/day/year or day/month/year date format and time values based on an A.M./P.M. clock, we use the date and time formats specified by the SQL Standard throughout the book. The three names used to refer to this data type are DATE, TIME, and TIMESTAMP. You can use the TIMESTAMP data type to store a combination of a date and time. Note that the names and usages for these data types vary depending on the database system you are using. Some systems store both date and time in the DATE data type, while others use TIMESTAMP or a data type called DATETIME. Consult your system documentation for details.
INTERVAL	This data type stores the quantity of time between two datetime values, expressed as either year, month; year/month; day, time; or day/time. Not all major database systems support the INTERVAL data type, so consult your system documentation for details.

Many database systems provide additional data types known as *extended data types* beyond those specified by the SQL Standard. (We listed a few of them in the previous list of data type categories.) Examples of extended data types include MONEY/CURRENCY and SERIAL/ROWID (for unique row identifiers).

Because our primary focus is on the *data manipulation* portion of SQL, you need be concerned only with the appropriate range of values for each data type your database system supports. This knowledge will help ensure that the expressions you define will execute properly, so be sure to familiarize yourself with the data types provided by your RDBMS program.

Changing Data Types: The CAST Function

You must be careful when you create an expression to make sure that the data types of the columns and literals are compatible with the operation you are requesting. For example, it doesn't make sense to try to add character data to a number. But if the character column or literal contains a number, you can use the CAST function to convert the value before trying to add another number. Figure 5–1 shows you the CAST function, which is supported in nearly all commercial database systems.

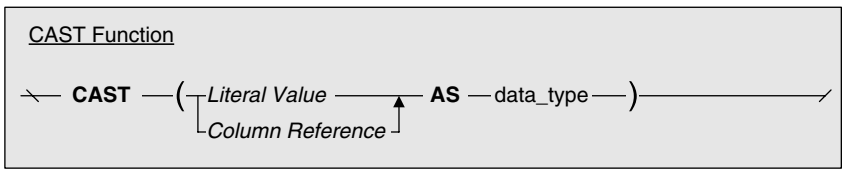


Figure 5–1 *The syntax diagram for the CAST function*

The CAST function converts a literal value or the value of a column into a specific data type. This helps to ensure that the data types of the values in the expression are *compatible*. By compatible we mean that all columns or literals in an expression are either characters, numbers, or datetime values. (As with any rule, there are exceptions that we'll mention later.) All the values you use in an expression must generally be compatible in order for the operation defined within the expression to work properly. Otherwise, your database system might raise an error message.

❖ **Note** Although most commercial database systems support the CAST function, some do not. Those systems that do not support CAST do have available a set of custom functions to achieve the same result. Consult your system documentation for details.

Converting a value in a column or a literal from one data type to another is a relatively intuitive and straightforward task. However, you'll have to keep the following restrictions in mind when you convert a value from its original data type to a different data type.

- Let's call this the "don't put a ten-pound sack in a five-pound box" rule. As mentioned earlier, you can define the maximum length of the data you want to store in a character data type. If you try to convert from one type of character field (for example, VARCHAR) to another character type (such as CHARACTER) and the data stored in the original column or literal is larger than the maximum length specified in the receiving data type, your database system will truncate the original character string. Your database system should also give you a warning that the truncation is about to occur.
- Let's call this the "don't put a square peg in a round hole" rule. You can convert a character column or literal to any other data type, but the character data in the source column or literal must be convertible to the target data type. For example, you can convert a five-character ZIP Code to a number, but you will encounter an error if your ZIP Code column contains Canadian postal codes that have letters. Note that the database system ignores any leading and/or trailing spaces when it converts a character column value to a numeric or datetime value. Also, most commercial systems support a wide range of character strings that are recognizable as date or time values. Consult your system documentation for details.
- This is the "ten-pound sack" rule, version 2. When you convert a numeric column's value to another numeric data type, the current contents of the convert-from column or literal had better fit in the target data type. For example, you will likely get an error if you attempt to convert a REAL value greater than 32,767 to a SMALLINT. Additionally, numbers to the right of the decimal place will be truncated or rounded as appropriate when you convert a number that has a decimal fraction to an INTEGER or SMALLINT. The amount of truncation or rounding is determined by the database system.
- But you can put "a square peg in a round hole" with certain limitations. When you convert the value of a numeric column to a character data type, one of three possible results will occur.
  1. It will convert successfully.
  2. Your system will pad it with blanks if its length is shorter than the defined length of the character column.
  3. The database system will raise an error if the character representation of the numeric value is longer than the defined length of the character column.

❖ **Note** Although the SQL Standard defines these restrictions, your database system might allow you some leeway when you convert a value from one data type to another. Some database systems provide automatic conversion for you without requiring you to use the CAST function. For example, some systems allow you to concatenate a number with text or to add text containing a number to another number without an explicit conversion. Refer to your database system's documentation for details.

It's important to note that this list does not constitute the entire set of restrictions defined by the SQL Standard. We listed only those restrictions that apply to the data types we use in this book. For a more in-depth discussion on data types and data conversion issues, please refer to any of the books listed in Appendix D, Suggested Reading.

Keep the CAST function in mind as you work through the rest of this book. You'll see us use it whenever appropriate to make sure we're working with compatible data types.

Specifying Explicit Values

The SQL Standard provides flexibility for enhancing the information returned from a SELECT statement by allowing use of constant values such as character strings, numbers, dates, times, or a suitable combination of these items, in any valid expression used within a SELECT statement. The SQL Standard categorizes these types of values as *literal values* and specifies the manner in which they are defined.

Character String Literals

A *character string literal* is a sequence of individual characters enclosed in *single* quotes. Yes, we know that you are probably used to using double quotes to enclose character strings, but we're presenting these concepts as the SQL Standard defines them. Figure 5–2 shows the diagram for a character string literal.

Here are a few examples of the types of character string literals you can define.

```
'This is a sample character string literal.'
'Here"s yet another!'
```

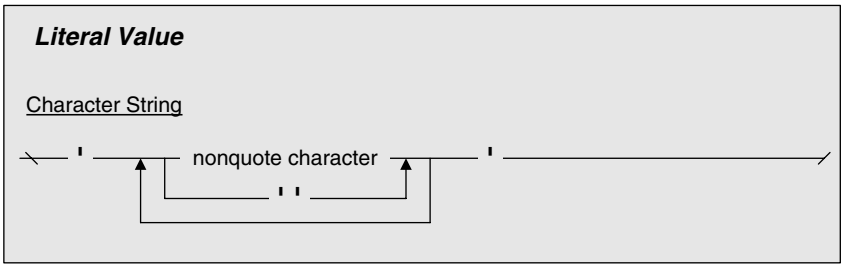


Figure 5–2 *The syntax diagram of a character string literal*

```
'B-28'
'Seattle'
```

You probably noticed what seemed to be a double quote in both the diagram and the second line of the previous example. Actually, it's not a double quote but two consecutive single quotes with no space between them. The SQL Standard states that a single quote embedded within a character string is represented by two consecutive single quotes. The SQL Standard defines it this way so that your database system can distinguish between a single quote that defines the beginning or end of a character string literal and a quote that you want included within the literal. The following two lines illustrate how this works.

```
SQL           'The Vendor''s name is: '
Displayed as  The Vendor's name is:
```

As we mentioned earlier, you can use character string literals to enhance the information returned by a SELECT statement. Although the information you see in a result set is usually easy to understand, it's very likely that the information can be made clearer. For example, if you execute the following SELECT statement, the result set displays only the vendor's Web site address and the vendor's name.

```
SQL         SELECT VendWebPage, VendName
            FROM Vendors
```

In some instances you can enhance the clarity of the information by defining a character string that provides supplementary descriptive text and then adding it to the SELECT clause. Use this technique judiciously because the character string literal will appear in each row of the result set. Here's how you might modify the previous example with a character string literal.

```
SQL         SELECT VendWebPage, 'is the Web site for',
               VendName
            FROM Vendors
```

A row in the result set generated by this SELECT statement looks like this.

www.viescas.com	is the Web site for	Viescas Consulting, Inc.

This somewhat clarifies the information displayed by the result set by identifying the actual purpose of the Web address. Although this is a simple example, it illustrates what you can do with character string literals. Later in this chapter, you'll see how you can use them in expressions.

❖ **Note** You'll find this technique especially useful when working with legacy databases that contain cryptic column names. However, you won't have to use this technique very often with your own databases if you follow the recommendations in Chapter 2, Ensuring Your Database Structure Is Sound.

Numeric Literals

A *numeric literal* is another type of literal you can use within a SELECT statement. As the name implies, it consists of an optional sign and a number and can include a decimal place, the exponent symbol, and an exponential number. Figure 5–3 shows the diagram for a numeric literal.

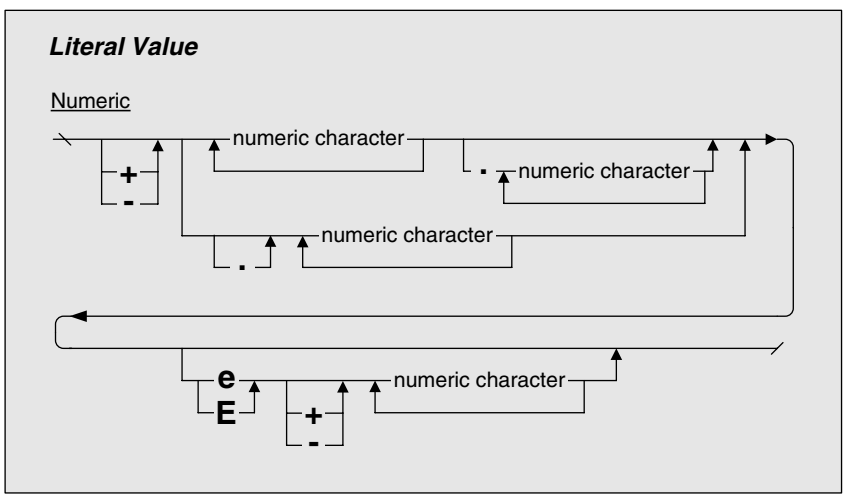


Figure 5–3 *The syntax diagram of a numeric literal*

Examples of numeric literals include the following:

```
427
-11.253
.554
0.3E-3
```

Numeric literals are most useful in expressions (for example, to multiply by or to add a fixed number value), so we'll postpone further discussion until later in this chapter.

Datetime Literals

You can supply specific dates and times for use within a SELECT statement by using *date literals*, *time literals*, and *timestamp literals*. The SQL Standard refers to these literals collectively as *datetime literals*. Defining these literals is a simple task, as Figure 5–4 shows.

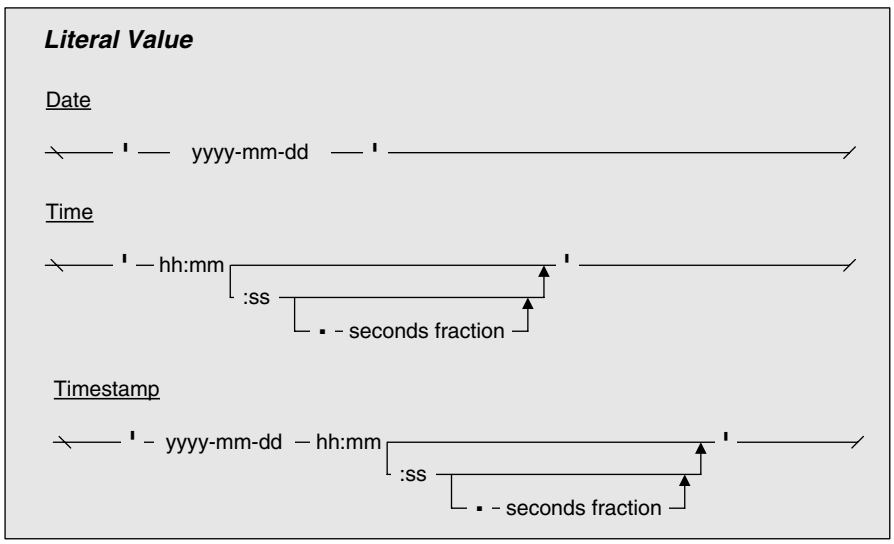


Figure 5–4 *The syntax diagram of date and time literals*

Bear in mind a few points, however, when using datetime and interval literals.

DATE — The format for a date literal is year-month-day, which is the format we follow throughout the book. However, many SQL databases allow the more common month/day/year format (United States) or day/month/year format (most non-U.S. countries). The SQL Standard also specifies that you include the DATE keyword before the literal, but nearly all commercial implementations allow you to simply specify the literal value surrounded by delimiter characters—usually single quotes. We found one case, the MySQL system, that requires you to specify a date literal in quotes and then to use the CAST function to convert the string to the DATE data type before you can use it in date calculations.

TIME — The hour format is based on a 24-hour clock. For example, 07:00 P.M. is represented as 19:00. The SQL Standard also specifies that you include the TIME keyword before the literal, but nearly all commercial implementations allow you to simply specify the literal value surrounded by delimiter characters—usually single quotes. We found one case, the MySQL system, that requires you to specify a time literal in quotes and then to use the CAST function to convert the string to the TIME data type before you can use it in time calculations.

TIMESTAMP — A timestamp literal is simply the combination of a date and a time separated by a single space. The rules for formatting the date and the time within a timestamp follow the individual rules for date and time. The SQL Standard also specifies that you include the TIMESTAMP keyword before the literal, but all commercial implementations that support the TIMESTAMP data type allow you to simply specify the literal value surrounded by delimiter characters—usually single quotes.

❖ **Note** In some systems, you can also define an *interval literal* to use in calculated expressions with datetime literals, but we won't be covering that type of literal in this book. See your system documentation for details.

You can find the diagrams for DATE, TIME, TIMESTAMP, and INTERVAL as defined by the SQL Standard in Appendix A, SQL Standard Diagrams.

Here are some examples of datetime literals.

```
'2007-05-16'
'2016-11-22'
'21:00'
```

```
'03:30:25'
‘2008-09-29 14:25:00'
```

Note that when using MySQL, you must explicitly convert any character literal containing a date or a time or a date and a time by using the CAST function. Here are some examples.

```
CAST('2016-11-22' AS DATE)
CAST('03:30:25' AS TIME)
CAST('2008-09-29 14:25:00' AS DATETIME)
```

As we noted previously, in order to follow the SQL Standard, you must precede each literal with a keyword indicating the desired value. Although the DATE and TIME keywords are defined in the SQL Standard as required components of date and time literals, respectively, most database systems rarely support these keywords in this particular context and require only the character string portion of the literal. Therefore, we'll refrain from using the keywords and instead use single quotes to delimit a date or time literal that appears in any example throughout the remainder of the book. We show you how to use dates and times in expressions later in this chapter. See Appendix A for more details on forming datetime literals that follow the SQL Standard.

Types of Expressions

You will generally use the following three types of expressions when working with SQL statements.

CONCATENATION	Combining two or more character columns or literals into a single character string
MATHEMATICAL	Adding, subtracting, multiplying, and dividing numeric columns or literals
DATE AND TIME ARITHMETIC	Applying addition or subtraction to dates and times

Concatenation

The SQL Standard defines two sequential vertical bars as the concatenation operator. You can concatenate two character items by placing a single item on either side of the concatenation operator. The result is a single string of

characters that is a combination of both items. Figure 5–5 shows the syntax diagram for the concatenation expression.

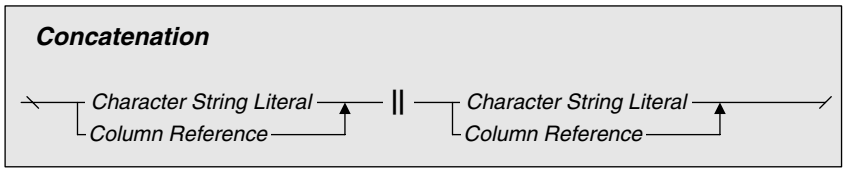


Figure 5–5 *The syntax diagram for the concatenation expression*

❖ **Note** Of the major database systems, we found that only IBM's DB2 and Informix and Oracle's Oracle support the SQL Standard operator for concatenation. Microsoft Office Access supports & and + as concatenation operators, Microsoft SQL Server and Ingres support +, and in MySQL you must use the CONCAT function. In all the examples in the book, we use the SQL Standard `||` operator. In the sample databases on the CD, we use the appropriate operator for each database type (Microsoft Access, Microsoft SQL Server, and MySQL).

Here's a general idea of how the concatenation operation works.

```
Expression    ItemOne || ItemTwo
Result        ItemOneItemTwo
```

Let's start with the easiest example in the world: concatenating two character string literals, such as a first name and a last name.

```
Expression    'Mike' || 'Hernandez'
Result        MikeHernandez
```

There are two points to consider in this example: First, single quotes are required around each name because they are character string literals. Second, the first and last names are right next to each other. Although the operation combined them correctly, it might not be what you expected. The solution is to add a space between the names by inserting another character literal that contains a single space.

```
Expression    'Mike' || ' ' || 'Hernandez'
Result        Mike Hernandez
```

The previous example shows that you can concatenate additional character values by using more concatenation operators. There is no limit to the number of character values you can concatenate, but there is a limit to the length of the character string the concatenation operation returns. In general, the length of the character string returned by a concatenation operation can be no greater than the maximum length allowed for a varying-length character data type. Your database system might handle this issue slightly differently, so check your documentation for further details.

Concatenating two or more character strings makes perfect sense, but you can also concatenate the values of two or more character columns in the same fashion. For example, suppose you have two columns called CompanyName and City. You can create an expression that concatenates the value of each column by using the column names within the expression. Here's an example that concatenates the values of both columns with a character string.

```
Expression    CompanyName || ' is based in ' || City
Result        DataTex Consulting Group is based in Seattle
```

You don't need to surround CompanyName or City with single quotes because they are column references. (Remember column references from the previous chapter?) You can use a column reference in any type of expression, as you'll see in the examples throughout the remainder of the book.

Notice that all the concatenation examples so far concatenate characters with characters. We suppose you might be wondering if you need to do anything special to concatenate a number or a date. Most database systems give you some leeway in this matter. When the system sees you trying to concatenate a character column or literal with a number or a date, the system automatically casts the data type of the number or date for you so that the concatenation works with compatible data types.

But you shouldn't always depend on your database system to quietly do the conversion for you. To concatenate a character string literal or the value of a character column with a date literal or the value of a numeric or date column, use the CAST function to convert the numeric or date value to a character string. Here's an example of using CAST to convert the value of a date column called DateEntered.

```
Expression  EntStageName || ' was signed with our agency on '
            || CAST(DateEntered as CHARACTER(10))
Result      Modern Dance was signed with our agency on 1995-05-16
```

❖ **Note** We specified an explicit length for the CHARACTER data type because the SQL Standard specifies that the absence of a length specification defaults to a length of 1. We found that most major implementations give you some leeway in this regard and generate a character string long enough to contain what you're converting. You can check your database documentation for details, but if you're in doubt, always specify an explicit length.

You should also use the CAST function to concatenate a numeric literal or the value of a numeric column to a character data type. In the next example, we use CAST to convert the value of a numeric column called RetailPrice.

```
Expression   ProductName || ' sells for ' || CAST(RetailPrice
             AS CHARACTER(8))
Result       Trek 9000 Mountain Bike sells for 1200.00
```

A concatenation expression can use character strings, datetime values, and numeric values simultaneously. The following example illustrates how you can use all three data types within the same expression.

```
Expression 'Order Number ' || CAST(OrderNumber AS CHARACTER(2))
           || ' was placed on ' ||
           CAST(OrderDate AS CHARACTER(10))
Result     Order Number 1 was placed on 2007-09-01
```

❖ **Note** The SQL Standard defines a variety of functions that you can use to extract information from a column or calculate a value across a range of rows. We'll cover some of these in more detail in Chapter 12, Simple Totals. Most commercial database systems also provide various functions to manipulate parts of strings or to format date, time, or currency values. Check your system documentation for details.

Now that we've shown how to concatenate data from various sources into a single character string, let's look at the different types of expressions you can create using numeric data.

Mathematical Expressions

The SQL Standard defines addition, subtraction, multiplication, and division as the operations you can perform on numeric data. Yes, we know—this is quite a limited set of operations! Fortunately, most RDBMS programs provide a much wider variety of operations, including modulus, square root, exponential, and absolute value. They also provide a wide array of scientific, trigonometrical, statistical, and mathematical functions as well. In this book, however, we focus only on those operations defined by the SQL Standard.

The order in which the four mathematical operations are performed—known as the *order of precedence*—is an important issue when you create mathematical expressions. The SQL Standard gives equal precedence to multiplication and division and specifies that they should be performed before any addition or subtraction. This is slightly contrary to the order of precedence you probably learned back in school, where multiplication is done before division, division before addition, and addition before subtraction, but it matches the order of precedence used in most modern programming languages. Mathematical expressions are evaluated from left to right. This could lead to some interesting results, depending on how you construct the expression! So, we strongly recommend that you make extensive use of parentheses in complex mathematical expressions to ensure that they evaluate properly.

If you remember how you created mathematical expressions back in school, then you already know how to create them in SQL. In essence, you use an optionally signed numeric value, a mathematical operator, and another optionally signed numeric value to create the expression. Figure 5-6 shows a diagram of this process.

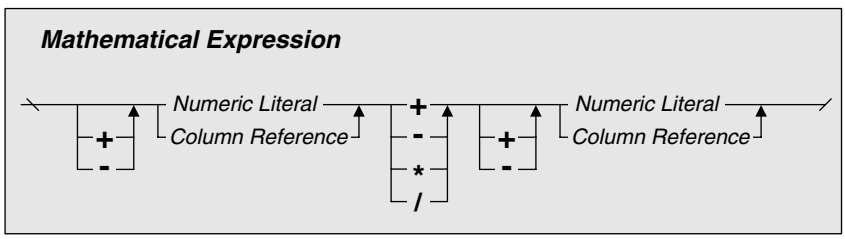


Figure 5–6 *The syntax diagram for a mathematical expression*

Here are some examples of mathematical expressions using numeric literal values, column references, and combinations of both.

```
25 + 35
-12 * 22
RetailPrice * QuantityOnHand
TotalScore / GamesBowled
RetailPrice - 2.50
TotalScore / 12
```

As mentioned earlier, you need to use parentheses to ensure that a complex mathematical expression evaluates properly. Here's a simple example of how you might use parentheses in such an expression.

```
Expression   (11 - 4) + (12 * 3)
Result       43
```

Pay close attention to the placement of parentheses in your expression because it affects the expression's resulting value. The two expressions in the following example illustrate this quite clearly. Although both expressions have the exact same numbers and operators, the placement of the parentheses is entirely different and causes the expressions to return completely different values.

```
Expression   (23 * 11) + 12
Result       265
Expression   23 * (11 + 12)
Result       529
```

It's easy to see why you need to be careful with parentheses, but don't let this stop you from using them. They are invaluable when working with complex expressions.

You can also use parentheses as a way to nest operations within an expression. When you use nested parenthetical operations, your database system evaluates them left to right and then in an "innermost to outermost" fashion. Here's an example of an expression that contains nested parenthetical operations.

```
Expression   (12 * (3 + 4)) - (24 / (10 + (6 - 4)))
Result       82
```

Executing the operations within the expression is not really as difficult as it seems. Here's the order in which your database system evaluates the expression.

1. (3 + 4) = 7
2. (12 * 7) = **84** *12 times the result of the first operation*
3. (6 – 4) = **2**
4. (10 + 2) = **12** *10 plus the result of the third operation*
5. (24 / 12) = **2** *24 divided by the result of the fourth operation*
6. 84 – 2 = **82** *84 minus the result of the second operation*

As you can see, the system proceeds left to right but must evaluate inner expressions when encountering an expression surrounded by parentheses. Effectively, (12 * (3 + 4)) and (24 / (10 + (6 – 4))) are on an equal level, so your system will completely evaluate the leftmost expression first, innermost to outermost. It then encounters the second expression surrounded by parentheses and evaluates that one innermost to outermost. The final operation subtracts from the result of the left expression the result of evaluating the right expression. (Does your head hurt yet? Ours do!)

Although we used numeric literals in the previous example, we could just as easily have used column references or a combination of numeric literals and column references as well. The key point to remember here is that you should plan and define your mathematical expressions carefully so that they return the results you seek. Use parentheses to clearly define the sequence in which you want operations to occur, and you'll get the result you expect.

When working with a mathematical expression, be sure that the values used in the expression are compatible. This is especially true of an expression that contains column references. You can use the CAST function for this purpose exactly as you did within a concatenation expression. For example, say you have a column called TotalLength based on an INTEGER data type that contains the whole number value 345, and a column called Distance based on a REAL data type that contains the value 138.65. To add the value of the Distance column to the value of the TotalLength column, you should use the CAST function to convert the Distance column's value into an INTEGER data type or the TotalLength column's value into a REAL data type, depending on whether you want the final result to be an INTEGER or a REAL data type. Assuming you're interested in adding only the integer values, you would accomplish this with the following expression.

```
Expression   TotalLength + CAST(Distance AS INTEGER)
Resulte      483
```

Not the answer you expected? Maybe you thought converting 138.65 to an integer would round the value up? Although the SQL Standard states that rounding during conversion using the CAST function depends on your database system, most systems truncate a value with decimal places when converting to an integer. So, we're assuming our system also does that and thus added 345 to 138, not the rounded value 139.

If you forget to ensure the compatibility of the column values within an expression, your database system might raise an error message. If it does, it will probably cancel the execution of the operations within the expression as well. Most RDBMS systems handle such conversions automatically without warning you, but they usually convert all numbers to the most complex data type before evaluating the expression. In the previous example, your RDBMS would most likely convert TotalLength to REAL (the more complex of the two data types). Your system will use REAL because all INTEGER values can be contained within the REAL data type. However, this might not be what you wanted. Those RDBMS programs that do not perform this sort of automatic conversion are usually good about letting you know that it's a data type mismatch problem, so you'll know what you need to do to fix your expression.

As you just learned, creating mathematical expressions is a relatively easy task as long as you do a little planning and know how to use the CAST function to your advantage. In our last discussion for this section, we'll show you how to create expressions that add and subtract dates and times.

Date and Time Arithmetic

The SQL Standard defines addition and subtraction as the operations you can perform on dates and times. Contrary to what you might expect, many RDBMS programs differ in the way they implement these operations. Some database systems allow you to define these operations as you would in a mathematical expression, while others require you to use special built-in functions for these tasks. Refer to your database system's documentation for details on how your particular RDBMS handles these operations. In this book, we discuss date and time expressions only in general terms so that we can give you an idea of how these operations should work.

Date Expressions

Figure 5–7 shows the syntax for a date expression as defined by the SQL Standard. As you can see, creating the expression is simple enough—take one value and add it to or subtract it from a second value.

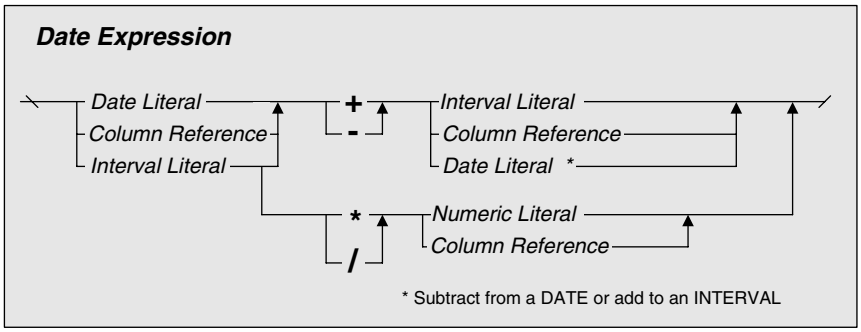


Figure 5–7 *The syntax diagram for a date expression*

The SQL Standard further defines the valid operations and their results as follows:

DATE plus or minus INTERVAL yields DATE
DATE minus DATE yields INTERVAL
INTERVAL plus DATE yields DATE
INTERVAL plus or minus INTERVAL yields INTERVAL
INTERVAL times or divided by NUMBER yields INTERVAL

Note that in the SQL Standard you can subtract only a DATE from a DATE or add only a DATE to an INTERVAL.

When you use a column reference, make certain it is based on a DATE or INTERVAL data type, as appropriate. If the column is not an acceptable data type, you might have to use the CAST function to convert the value you are adding or subtracting. The SQL Standard explicitly specifies that you can perform these operations only using the indicated data types, but many database systems convert the column's data type for you automatically. Your RDBMS will ultimately determine whether the conversion is required, so check your documentation.

Although only a few commercial systems support the INTERVAL data type, nearly all of them allow you to use an integer value (such as SMALLINT or INT) to add to or subtract from a date value. You can think of this as adding and subtracting days. This allows you to answer questions such as *"What is the date nine days from now?"* and *"What was the date five days ago?"* Note also that some database systems allow you to add to or subtract from a datetime value using a fraction. For example, adding 3.5 to a datetime value in Microsoft Access adds 3 days and 12 hours.

When you subtract a date from another date, you are calculating the interval between the two dates. For example, you might need to subtract a hire date from the current date to determine how long an employee has been with the company. Although the SQL Standard indicates that you can add only an interval to a date, many database systems (especially those that do not support the INTERVAL data type) allow you to add either a number or a date anyway. You can use this sort of calculation to answer questions such as *"When is the employee's next review date?"*

In this book, we'll show you simple calculations using dates and assume that you can at least add an integer number of days to a date value. We'll also assume that subtracting one date from another yields an integer number of days between the two dates. If you apply these simple concepts, you can create most of the date expressions that you'll need. Here are some examples of the types of date expressions you can define.

```
'2007-05-16' - 5
'2007-11-14' + 12
ReviewDate + 90
EstimateDate - DaysRequired
'2007-07-22' - '2007-06-13'
ShipDate - OrderDate
```

Time Expressions

You can create expressions using time values as well, and Figure 5–8 shows the syntax. Date and time expressions are very similar, and the same rules and restrictions that apply to a date expression also apply to a time expression.

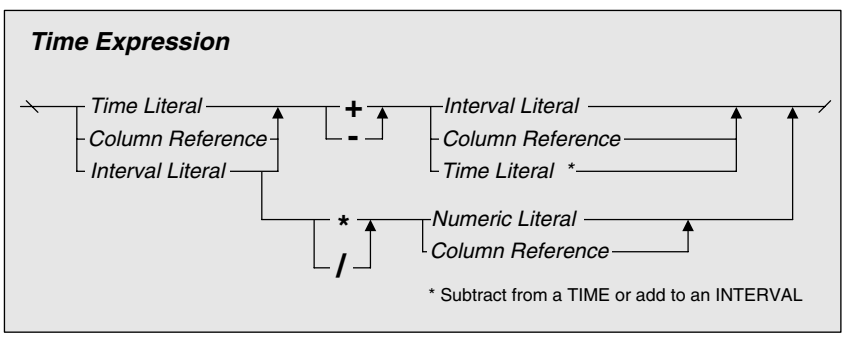


Figure 5–8 *The syntax diagram for a time expression*

The SQL Standard further defines the valid operations and their results as follows:

TIME plus or minus INTERVAL yields TIME
TIME minus TIME yields INTERVAL
INTERVAL plus or minus INTERVAL yields INTERVAL
INTERVAL times or divided by NUMBER yields INTERVAL

Note that in the SQL Standard you can subtract only a TIME from a TIME or add only a TIME to an INTERVAL.

All the same "gotchas" we noted for date expressions apply to time expressions. In addition, for systems that support a combination datetime data type, the time portion of the value is stored as a fraction of a day accurate at least to seconds. When using systems that support datetime, you can also usually add or subtract a decimal fraction value to a datetime value. For example, 0.25 is six hours (one-fourth of a day). In this book, we'll assume that your system supports both adding and subtracting time literals or columns. We make no assumption about adding or subtracting decimal fractions. Again, check your documentation to find out what your system actually supports.

Given our assumptions, here are some general examples of time expressions.

```
'14:00' + '00:22'
'19:00' - '16:30'
StartTime + '00:19'
StopTime - StartTime
```

We said earlier that we would present date and time expressions only in general terms. Our goal was to make sure that you understood date and time expressions conceptually and that you had a general idea of the types of expressions you should be able to create. Unfortunately, most database systems do not implement the SQL Standard's specification for time expressions exactly, and many only partially support the specification for the date expression. As we noted, however, all database systems provide one or more functions that allow you to work with dates and times. You can find a summary of these functions for five major implementations in Appendix C, Date and Time Functions. We strongly recommend that you study your database system's documentation to learn what types of functions your system provides.

Now that you know how to create the various types of expressions, the next step is to learn how to use them.

Using Expressions in a SELECT Clause

Knowing how to use expressions is arguably one of the most important concepts you'll learn in this book. You'll use expressions for a variety of purposes when working with SQL. For example, you would use an expression to

- Create a calculated column in a query
- Search for a specific column value
- Filter the rows in a result set
- Connect two tables in a JOIN operation

We'll show you how to do this (and more) as we work through the rest of the book. We begin by showing you how to use basic expressions in a SELECT clause.

❖ **Note** Throughout this chapter, we use the "Request/Translation/Clean Up/SQL" technique introduced in Chapter 4.

You can use basic expressions in a SELECT clause to clarify information in a result set and to expand the result set's scope of information. For example, you can create expressions to concatenate first and last names, calculate the total price of a product, determine how long it took to complete a project, or specify a date for a patient's next appointment. Let's look at how you might use a concatenation expression, a mathematical expression, and a date expression in a SELECT clause. First, we'll work with the concatenation expression.

Working with a Concatenation Expression

Unlike mathematical and date expressions, you use concatenation expressions only to enhance the readability of the information contained in the result set of a SELECT statement. Suppose you are posing the following request:

"Show me a current list of our employees and their phone numbers."

When translating this request into a SELECT statement, you can improve the output of the result set somewhat by concatenating the first and last names into a single column. Here's one way you can translate this request.

Translation	Select the first name, last name, and phone number of all our employees from the employees table
Clean Up	Select ~~the~~ first name, last name, ~~and~~ phone number ~~of all our employees~~ from ~~the~~ employees ~~table~~

SQL	`SELECT EmpFirstName || ' ' || EmpLastName,`
	`  'Phone Number: ' || EmpPhoneNumber`
	`FROM Employees`

The result for one of the rows will look something like this.

Mary Thompson	Phone Number: 555-2516

You probably noticed that in addition to concatenating the first name column, a space, and the last name column, we also concatenated the character literal string "Phone Number: " with the phone number column. This example clearly shows that you can easily use more than one concatenation expression in a SELECT clause to enhance the readability of the information in the result set. Remember that you can also concatenate values with different data types by using the CAST function. For instance, we concatenate a character column value with a numeric column value in the next example.

"Show me a list of all our vendors and their identification numbers."

Translation	Select the vendor name and vendor ID from the vendors table
Clean Up	Select ~~the~~ vendor name ~~and~~ vendor ID from ~~the~~ vendors ~~table~~

SQL	`SELECT 'The ID Number for ' || VendName ||`
	`  ' is ' || CAST(VendorID AS CHARACTER)`
	`FROM Vendors`

Although the concatenation expression is a useful tool in a SELECT statement, it is one that you should use judiciously. When you use concatenation expressions containing long character string literals, keep in mind that the literals will appear in every row of the result set. You might end up cluttering the final result with repetitive information instead of enhancing it. Carefully consider your use of literals in concatenation expressions so that they work to your advantage.

Naming the Expression

When you use an expression in a SELECT clause, the result set includes a new column that displays the result of the operation defined in the expression. This new column is known as a calculated (or derived) column. For example,

the result set for the following SELECT statement will contain three columns—two "real" columns and one calculated column.

```
SQL         SELECT EmpFirstName || ' ' || EmpLastName,
               EmpPhoneNumber, EmpCity
            FROM Employees
```

The two real columns are, of course, EmpPhoneNumber and EmpCity, and the calculated column is derived from the concatenation expression at the beginning of the SELECT clause.

According to the SQL Standard, you can optionally provide a name for the new column by using the AS keyword. (In fact, you can assign a new name to any column using the AS clause.) Almost every database system, however, *requires* a name for a calculated column. Some database systems require you to provide the name explicitly, while others actually provide a generated name for you. Determine how your database system handles this before you work with the examples. If you plan to reference the result of the expression in your query, you should provide a name.

Figure 5–9 shows the syntax for naming an expression. You can use any valid character string literal (enclosed in single quotes) for the name. Some database systems relax this requirement when you're naming an expression and require quotes only when your column name includes embedded spaces. However, we strongly recommend that you not use spaces in your names because the spaces can be difficult to deal with in some database programming languages.

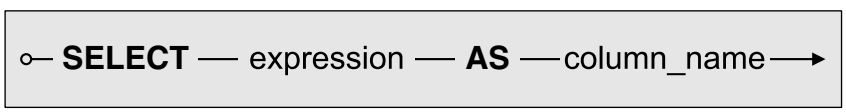


Figure 5–9 *The syntax diagram for naming an expression*

Now we'll modify the SELECT statement in the previous example and supply a name for the concatenation expression.

```
SQL         SELECT EmpFirstName || ' ' || EmpLastName AS
               EmployeeName, EmpPhoneNumber, EmpCity
            FROM Employees
```

The result set for this SELECT statement will now contain three columns called EmployeeName, EmpPhoneNumber, and EmpCity.

In addition to supplying a name for expressions, you can use the AS keyword to supply an alias for a real column name. Suppose you have a column called DOB and are concerned that some of your users might not be familiar with this abbreviation. You can eliminate any possible misinterpretation of the name by using an alias, as shown here.

```
SQL         SELECT EmpFirstName || ' ' || EmpLastName AS
               EmployeeName, DOB AS DateOfBirth
            FROM Employees
```

This SELECT statement produces a result set with two columns called EmployeeName and DateOfBirth. You've now effectively eliminated any possible confusion of the information displayed in the result set.

Providing names for your calculated columns has a minor effect on the translation process. For example, here's one possible version of the translation process for the previous example.

"Give me a list of employee names and their dates of birth."

Translation	Select first name and last name as employee name and DOB as date of birth from the employees table
Clean Up	Select first name ~~and~~ `||` `''` `||` last name as EmployeeName ~~and~~ DOB as DateOfBirth from ~~the~~ employees ~~table~~
SQL	`SELECT EmpFirstName || ' ' || EmpLastName AS EmployeeName, DOB AS DateOfBirth FROM Employees`

After you become accustomed to using expressions, you won't need to state them quite as explicitly in your translation statements as we did here. You'll eventually be able to easily identify and define the expressions you need as you construct the SELECT statement itself.

❖ **Note** Throughout the remainder of the book, we provide names for all calculated columns within an SQL statement, as appropriate.

Working with a Mathematical Expression

Mathematical expressions are possibly the most versatile of the three types of expressions, and you'll probably use them quite often. For example, you can use a mathematical expression to calculate a line item total, determine the

average score from a given set of tests, calculate the difference between two lab results, and estimate the total seating capacity of a building. The real trick is to make certain your expression works, and that is just a function of doing a little careful planning.

Here's an example of how you might use a mathematical expression in a SELECT statement.

"Display for each agent the agent name and projected income (salary plus commission), assuming each agent will sell $50,000 worth of bookings."

Translation: Select first name and last name as agent name and salary plus 50000 times commission rate as projected income from the agents table

Clean Up: Select first name ~~and~~ || ' ' || last name as AgentName, ~~and~~ salary ~~plus~~ + 50000 ~~times~~ * commission rate as Projected Income from ~~the~~ agents ~~table~~

SQL:
```
SELECT AgtFirstName || ' ' || AgtLastName
   AS AgentName, Salary + (50000 * CommissionRate)
   AS ProjectedIncome
FROM Agents
```

Notice that we added parentheses to make it crystal clear that we expect the commission rate to be multiplied by 50,000 and then add the salary, not add 50,000 to the salary and then multiply by the commission rate. As the example shows, you're not limited to using a single type of expression in a SELECT statement. Rather, you can use a variety of expressions to retrieve the information you need in the result set. Here's another way you can write the previous SQL statement.

```
SQL     SELECT AgtFirstName || ' ' || AgtLastName
           || ' has a projected income of ' ||
           CAST(Salary + (50000 * CommissionRate) AS CHARACTER)
           AS ProjectedIncome
        FROM Agents
```

The information you can provide by using mathematical expressions is virtually limitless, but you must properly plan your expressions and use the CAST function as appropriate.

Working with a Date Expression

Using a date expression is similar to using a mathematical expression in that you're simply adding or subtracting values. You can use date expressions for

all sorts of tasks. For example, you can calculate an estimated ship date, project the number of days it will take to finish a project, or determine a follow-up appointment date for a patient. Here's an example of how you might use a date expression in a SELECT clause.

"How many days did it take to ship each order?"

Translation	Select the order number and ship date minus order date as days to ship from the orders table
Clean Up	Select ~~the~~ order number ~~and~~ ship date ~~minus~~ – order date as DaysToShip from ~~the~~ orders ~~table~~
SQL	`SELECT OrderNumber, CAST(ShipDate – OrderDate AS INTEGER) AS DaysToShip` `FROM Orders`

You can use time expressions in the same manner.

"What would be the start time for each class if we began each class ten minutes later than the current start time?"

Translation	Select the start time and start time plus 10 minutes as new start time from the classes table
Clean Up	Select ~~the~~ start time ~~and~~ start time ~~plus~~ + '00:10' ~~minutes~~ as NewStartTime from ~~the~~ classes ~~table~~
SQL	`SELECT StartTime, StartTime + '00:10' AS NewStartTime` `FROM Classes`

As we mentioned earlier, all database systems provide a function or set of functions for working with date values. We did want to give you an idea of how you might use dates and times in your SELECT statements, however, and we again recommend that you refer to your database system's documentation for details on the date and time functions your database system provides.

A Brief Digression: Value Expressions

You now know how to use column references, literal values, and expressions in a SELECT clause. You also know how to assign a name to a column reference or an expression. Now we'll show you how this all fits into the larger scheme of things.

The SQL Standard refers to a column reference, literal value, and expression collectively as a *value expression*. Figure 5–10 shows how to define a value expression.

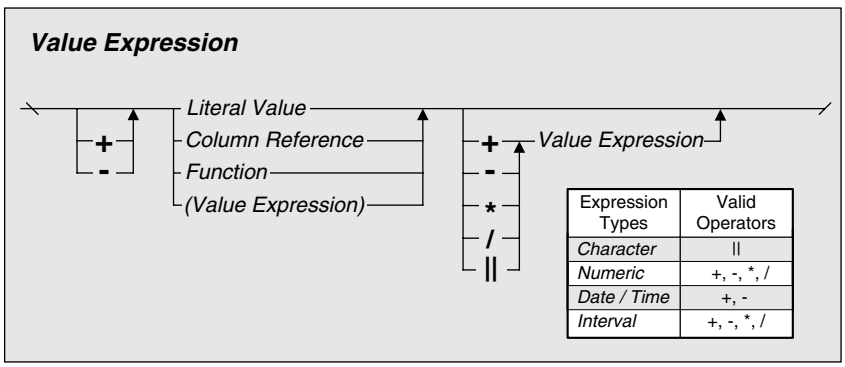


Figure 5–10 *The syntax diagram for a value expression*

Let's take a closer look at the components of a value expression.

- The syntax begins with an optional plus or minus sign. You use either of these signs when you want the value expression to return a signed numeric value. The value itself can be a numeric literal, the value of a numeric column, a call to a function that returns a numeric value (see our discussion of the CAST function earlier in this chapter), or the return value of a mathematical expression. You cannot use the plus or minus sign before an expression that returns a character data type.
- You can see that the first list in the figure also includes "(*Value Expression*)." This means that you can use a complex value expression comprised of other value expressions that include concatenation or mathematical operators of their own. The parentheses force the database system to evaluate this value expression first.
- The next item in the syntax is a list of operators. As you can see in the inset box, the type of expression you use at the beginning of the syntax determines which operators you can select from this list.
- No, you're not seeing things: "*Value Expression*" does appear after the list of operators as well. The fact that you can use other value expressions within a value expression allows you to create very complex expressions.

By its very definition, a value expression returns a value that is used by some component of an SQL statement. The SQL Standard specifies the use of a value expression in a variety of statements and defined terms. No matter where you use it, you'll always define a value expression in the same manner as you've learned here.

We'll put this all into some perspective by showing you how a value expression is used in a SELECT statement. Figure 5–11 shows a modified version of the SELECT statement syntax diagram presented in Figure 4–9 in Chapter 4. This new syntax gives you the flexibility to use literals, column references, expressions, or any combination of these within a single SELECT statement. You can optionally name your value expressions with the AS keyword.

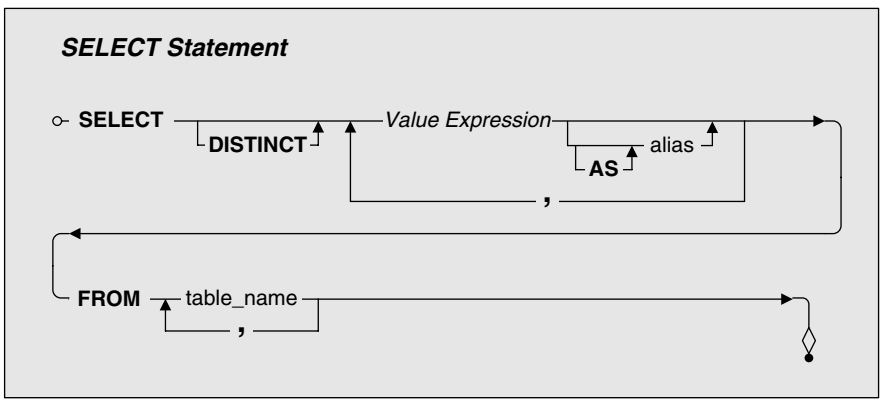


Figure 5–11 *The syntax diagram for the SELECT statement that includes a value expression*

Throughout the remainder of the book, we use the term *value expression* to refer to a column reference, a literal value, or an expression, as appropriate. In later chapters, we discuss how to use a value expression in other statements and show you a couple of other items that a value expression represents.

Now, back to our regularly scheduled program.

That "Nothing" Value: Null

As you know, a table consists of columns and rows. Each column represents a characteristic of the subject of the table, and each row represents a unique instance of the table's subject. You can also think of a row as one complete

set of column values—each row contains exactly one value from each column in the table. Figure 5–12 shows an example of a typical table.

Customers

CustomerID	CustFirstName	CustLastName	CustStreetAddress	CustCity	CustCounty	CustState
1001	Suzanne	Viescas	15127 NE 24th, #383	Redmond	King	WA
1002	William	Thompson	122 Spring River Drive	Duvall	King	WA
1003	Gary	Hallmark	Route 2, Box 203B	Auburn	King	WA
1004	Robert	Brown	672 Lamont Ave	Houston		TX
1005	Dean	McCrae	4110 Old Redmond Rd.	Redmond		WA
1006	John	Viescas	15127 NE 24th, #383	Redmond	King	WA
1007	Mariya	Sergienko	901 Pine Avenue	Portland		OR
1008	Neil	Patterson	233 West Valley Hwy	San Diego	San Diego	CA

Figure 5–12 *A typical Customers table*

So far we've shown how to retrieve information from the data in a table with a SELECT statement and how to manipulate that data by using value expressions. All of this works just fine because we've continually made the assumption that each column in the table contains data. But as Figure 5–12 clearly illustrates, a column sometimes might not contain a value for a particular row in the table. Depending on how you use the data, the absence of a value might adversely affect your SELECT statements and value expressions. Before we discuss any implications, let's first examine how SQL regards missing values.

Introducing Null

In SQL, a *Null* represents a *missing* or an *unknown* value. You must understand from the outset that a Null *does not* represent a zero, a character string of one or more blank spaces, or a "zero-length" character string. The reasons are quite simple.

- A zero can have a very wide variety of meanings. It can represent the state of an account balance, the current number of available first-class ticket upgrades, or the current stock level of a particular product.
- Although a character string of one or more blank spaces is guaranteed to be meaningless to most of us, it is something that is definitely meaningful to SQL. A blank space is a valid character as far as SQL is concerned, and a character string composed of three blank spaces (`'   '`) is just as legitimate as a character string composed of several letters (`'a character string'`).

- A zero-length string—two consecutive single quotes with no space in between (`''`)—can be meaningful under certain circumstances. In an employee table, for example, a zero-length string value in a column called MiddleInitial might represent the fact that a particular employee does not have a middle initial in her name. Note, however, that some implementations (notably Oracle) treat a zero-length string in a VARCHAR as Null.

A Null is quite useful when used for its stated purpose, and the Customers table in Figure 5–12 shows a clear example of this. In the CustCounty column, each blank cell represents a missing or unknown county name for the row in which it appears—a Null. In order to use Nulls correctly, you must understand why they occur in the first place.

Missing values are commonly the result of human error. Consider the row for Robert Brown, for example. If you're entering the data for Mr. Brown and you fail to ask him for the name of the county he lives in, that data is considered missing and is represented in the row as a Null. After you recognize the error, however, you can correct it by calling Mr. Brown and asking him for the county name.

Unknown values appear in a table for a variety of reasons. One reason might be that a specific value you need for a column is as yet undefined. For example, you might have a Categories table in a School Scheduling database that doesn't have a category for a new set of classes that you want to offer beginning in the fall session. Another reason a table might contain unknown values is that the values are truly unknown. Let's use the Customers table in Figure 5–12 once again and consider the row for Dean McCrae. Say that you're entering the data for Mr. McCrae, and you ask him for the name of the county he lives in. If neither of you knows the county that includes the city in which he lives, then the value for the county column in his row is truly unknown. This is represented in his row as a Null. Obviously, you can correct the problem after either of you determines the correct county name.

A column value might also be Null if none of its values apply to a particular row. Let's assume for a moment that you're working with an employee table that contains a Salary column and an HourlyRate column. The value for one of these two columns is always going to be Null because an employee cannot be paid both a fixed salary and an hourly rate.

It's important to note that there is a very slim difference between "does not apply" and "is not applicable." In the previous example, the value of one of the

two columns literally does not apply. But let's assume you're working with a patient table that contains a column called HairColor and you're currently updating a row for an existing male patient. If that patient is bald, then the value for that column is definitely not applicable. Although you could just use a Null to represent a value that is not applicable, we recommend that you use a true value such as "N/A" or "Not Applicable." This will make the information clearer in the long run.

As you can see, whether you allow Nulls in a table depends on the manner in which you're using the data. Now that we've shown you the positive side of using Nulls, let's take a look at the negative implication of using Nulls.

The Problem with Nulls

The major drawback of Nulls is their adverse effect on mathematical operations. Any operation involving a Null evaluates to Null. This is logically reasonable—if a number is unknown, then the result of the operation is necessarily unknown. Note how a Null alters the outcome of the operation in the next example.

```
(25 * 3) + 4 = 79
(Null * 3) + 4 = Null
(25 * Null) + 4 = Null
(25 *3) + Null = Null
```

The same result occurs when an operation involves columns containing Null values. For example, suppose you execute the following SELECT statement and it returns the result set shown in Figure 5–13.

```
SQL        SELECT ProductID, ProductDescription, Category,
              Price, QuantityOnHand, Price *
              QuantityOnHand AS TotalValue
           FROM Products
```

The operation represented by the TotalValue column is completed successfully as long as both the Price and QuantityOnHand columns have valid numeric values. Otherwise, TotalValue will contain a Null if either Price or QuantityOnHand contains a Null. The good news is that TotalValue will contain an appropriate value after you replace the Nulls in Price and QuantityOnHand with valid numeric values. You can avoid this problem completely by ensuring that the columns you use in a mathematical expression do not contain Null values.

ProductID	ProductDescription	Category	Price	QuantityOnHand	TotalValue
70001	Shur-Lok U-Lock	Accessories		12	
70002	SpeedRite Cyclecomputer		65.00	20	1,300.00
70003	SteelHead Microshell Helmet	Accessories	36.00	33	1,118.00
70004	SureStop 133-MB Brakes	Components	23.50	16	376.00
70005	Diablo ATM Mountain Bike	Bikes	1,200.00		
70006	UltraVision Helmet Mount Mirrors		7.45	10	74.50

Figure 5–13 *Nulls involved in a mathematical expression*

This is not the only time we'll be concerned with Nulls. In Chapter 12, we'll see how Nulls impact SELECT statements that summarize information.

Sample Statements

Now that you know how to use various types of value expressions in the SELECT clause of a SELECT statement, let's take a look, on the next few pages, at some examples using the tables from four of the sample databases. These examples illustrate the use of expressions to generate an output column.

We've also included sample result sets that would be returned by these operations and placed them immediately after the SQL syntax line. The name that appears immediately above a result set is the name we gave each query in the sample data on the companion CD you'll find bound into the back of the book. We stored each query in the appropriate sample database (as indicated within the example) and prefixed the names of the queries relevant to this chapter with "CH05." You can follow the instructions in the Introduction of this book to load the samples onto your computer and try them.

❖ **Note** We've combined the Translation and Clean Up steps in the following examples so that you can begin to learn how to consolidate the process. Although you'll still work with all three steps during the body of any given chapter, you'll get a chance to work with the consolidated process in each Sample Statements section.

Sales Orders Database

"What is the inventory value of each product?"

Translation/ Clean Up: Select ~~the~~ product name, retail price ~~times~~ * quantity on hand as InventoryValue from ~~the~~ products ~~table~~

SQL:

```
SELECT ProductName,
   RetailPrice * QuantityOnHand AS
   InventoryValue
FROM Products
```

CH05_Product_Inventory_Value (40 Rows)

ProductName	InventoryValue
Trek 9000 Mountain Bike	$7,200.00
Eagle FS-3 Mountain Bike	$14,400.00
Dog Ear Cyclecomputer	$1,500.00
Victoria Pro All Weather Tires	$1,099.00
Dog Ear Helmet Mount Mirrors	$89.40
Viscount Mountain Bike	$3,175.00
Viscount C-500 Wireless Bike Computer	$1,470.00
Kryptonite Advanced 2000 U-Lock	$1,000.00
<< more rows here >>	

"How many days elapsed between the order date and the ship date for each order?"

Translation/ Clean Up: Select ~~the~~ order number, order date, ship date, ship date ~~minus~~ - order date as DaysElapsed from ~~the~~ orders ~~table~~

SQL:

```
SELECT OrderNumber, OrderDate, ShipDate,
   CAST(ShipDate - OrderDate AS INTEGER)
   AS DaysElapsed
FROM Orders
```

CH05_Shipping_Days_Analysis (944 Rows)

OrderNumber	OrderDate	ShipDate	DaysElapsed
1	2007-09-01	2007-09-04	3
2	2007-09-01	2007-09-03	2
3	2007-09-01	2007-09-04	3
4	2007-09-01	2007-09-03	2
5	2007-09-01	2007-09-01	0
6	2007-09-01	2007-09-05	4
7	2007-09-01	2007-09-04	3
8	2007-09-01	2007-09-01	0
9	2007-09-01	2007-09-04	3
10	2007-09-01	2007-09-04	3
<< more rows here >>			

Entertainment Agency Database

"How long is each engagement due to run?"

Translation/ Clean Up: Select ~~the~~ engagement number, end date ~~minus~~ – start date ~~plus one~~ + 1 as DueToRun from ~~the~~ engagements ~~table~~

SQL:

```
SELECT EngagementNumber,
   CAST(CAST(EndDate - StartDate AS INTEGER) + 1
   AS CHARACTER)
   || ' day(s)' AS DueToRun
FROM Engagements
```

CH05_Engagement_Lengths (111 Rows)

EngagementNumber	DueToRun
2	5 day(s)
3	6 day(s)
4	7 day(s)
5	4 day(s)
6	5 day(s)
7	8 day(s)
8	8 day(s)
9	11 day(s)
10	10 day(s)
11	2 day(s)
<< more rows here >>	

❖ **Note** You have to add "1" to the date expression in order to account for each date in the engagement. Otherwise, you'll get "0 day(s)" for an engagement that starts and ends on the same date. You can also see that we CAST the result of subtracting the two dates first as INTEGER so that we could add the value 1, then CAST the result of that to CHARACTER to ensure the concatenation works as expected.

"What is the net amount for each of our contracts?"

Translation/ Clean Up: Select ~~the~~ engagement number, contract price, contract price ~~times~~ * 0.12 as OurFee, contract price ~~minus~~ – (contract price ~~times~~ * 0.12) as NetAmount from ~~the~~ engagements ~~table~~

SQL:

```
SELECT EngagementNumber, ContractPrice,
   ContractPrice * 0.12 AS OurFee,
   ContractPrice -(ContractPrice * 0.12)
   AS NetAmount
FROM Engagements
```

CH05_Net_Amount_Per_Contract (111 Rows)

EngagementNumber	ContractPrice	OurFee	NetAmount
2	$200.00	$24.00	$176.00
3	$590.00	$70.80	$519.20
4	$470.00	$56.40	$413.60
5	$1,130.00	$135.60	$994.40
6	$2,300.00	$276.00	$2,024.00
7	$770.00	$92.40	$677.60
8	$1,850.00	$222.00	$1,628.00
9	$1,370.00	$164.40	$1,205.60
10	$3,650.00	$438.00	$3,212.00
11	$950.00	$114.00	$836.00
<< more rows here >>			

School Scheduling Database

"List how many years each staff member has been with the school as of October 1, 2007, and sort the result by last name and first name."

Translation/ Clean Up: Select last name || ', ' || ~~and~~ first name ~~concatenated with a comma~~ as Staff, date hired, ~~and~~ (('2007-10-01' ~~minus~~ – date hired) ~~divided by~~ / 365) as YearsWithSchool from ~~the~~ staff ~~table and sort~~ order by last name ~~and~~ first name

SQL:

```
SELECT StfLastName || ', ' || StfFirstName AS Staff,
   DateHired,
   CAST(CAST('2007-10-01' - DateHired AS INTEGER)
   / 365 AS INTEGER)
   AS YearsWithSchool
FROM Staff
ORDER BY StfLastName, StfFirstName
```

CH05_Length_Of_Service (27 Rows)

Staff	DateHired	YearsWithSchool
Alborous, Sam	1982-11-20	25
Black, Alastair	1988-12-11	19
Bonnicksen, Joyce	1986-03-02	22
Brehm, Peter	1986-07-16	21
Brown, Robert	1989-02-09	19
Coie, Caroline	1983-01-28	25
DeGrasse, Kirk	1988-03-02	20
Ehrlich, Katherine	1985-03-08	23
Glynn, Jim	1985-08-02	22
Hallmark, Alaina	1984-01-07	24
<< more rows here >>		

❖ **Note** The expression in this SELECT statement is technically correct and works as expected, but it returns the wrong answer for any leap year. You can correct this problem by using the appropriate date arithmetic function provided by your database system. As mentioned earlier, most database systems provide their own methods of working with dates and times.

"Show me a list of staff members, their salaries, and a proposed 7 percent bonus for each staff member."

Translation/ Clean Up: Select ~~the~~ last name || ', ' || ~~and~~ first name as StaffMember, salary, ~~and~~ salary ~~times~~ * 0.07 as Bonus from ~~the~~ staff ~~table~~

SQL:

```
SELECT StfLastName || ', ' || StfFirstName
   AS Staff, Salary, Salary * 0.07 AS Bonus
FROM Staff
```

CH05_Proposed_Bonuses (27 Rows)

Staff	Salary	Bonus
Alborous, Sam	$60,000.00	$4,200.00
Black, Alastair	$60,000.00	$4,200.00
Bonnicksen, Joyce	$60,000.00	$4,200.00
Brehm, Peter	$60,000.00	$4,200.00
Brown, Robert	$49,000.00	$3,430.00
Coie, Caroline	$52,000.00	$3,640.00
DeGrasse, Kirk	$45,000.00	$3,150.00
Ehrlich, Katherine	$45,000.00	$3,150.00
Glynn, Jim	$45,000.00	$3,150.00
Hallmark, Alaina	$57,000.00	$39,900.00
<< more rows here >>		

Bowling League Database

"Display a list of all bowlers and addresses formatted suitably for a mailing list, sorted by ZIP Code."

Translation/ Clean Up: Select first name || ' ' || ~~and~~ last name as FullName, BowlerAddress, city || ', ' || state || ' ' || ~~and~~ ZIP Code as CityStateZip from ~~the~~ bowlers ~~table and~~ order by ZIP Code

SQL:

```
SELECT BowlerFirstName || ' ' || BowlerLastName AS
FullName,
   Bowlers.BowlerAddress,
   BowlerCity || ', ' || BowlerState || ' ' ||
   BowlerZip AS
   CityStateZip
FROM Bowlers
ORDER BY BowlerZip
```

CH05_Names_Addresses_For_Mailing (32 Rows)

FullName	BowlerAddress	CityStateZip
Kathryn Patterson	16 Maple Lane	Auburn, WA 98002
Rachel Patterson	16 Maple Lane	Auburn, WA 98002
Ann Patterson	16 Maple Lane	Auburn, WA 98002
Neil Patterson	16 Maple Lane	Auburn, WA 98002
Megan Patterson	16 Maple Lane	Auburn, WA 980025
Carol Viescas	16345 NE 32nd Street	Bellevue, WA 98004
Sara Sheskey	17950 N 59th	Seattle, WA 98011
Richard Sheskey	17950 N 59th	Seattle, WA 98011
William Thompson	122 Spring Valley Drive	Duvall, WA 98019
Mary Thompson	122 Spring Valley Drive	Duvall, WA 98019
<< more rows here >>		

"What was the point spread between a bowler's handicap and raw score for each match and game played?"

Translation/ Clean Up: Select bowler ID, match ID, game number, handicap score, raw score, handicap score ~~minus~~ – raw score as PointDifference from ~~the~~ bowler scores ~~table and~~ order by bowler ID, match ID, game number

SQL:

```
SELECT BowlerID, MatchID, GameNumber,
   HandiCapScore, RawScore,
   HandiCapScore - RawScore AS PointDifference
FROM Bowler_Scores
ORDER BY BowlerID, MatchID, GameNumber
```

CH05_Handicap_vs_RawScore (1344 Rows)

BowlerID	MatchID	GameNumber	HandiCapScore	RawScore	PointDifference
1	1	1	192	146	46
1	1	2	192	146	46
1	1	3	199	153	46
1	5	1	192	145	47
1	5	2	184	137	47
1	5	3	199	152	47
1	10	1	189	140	49
1	10	2	186	137	49
1	10	3	210	161	49
<< more rows here >>					

SUMMARY

We began the chapter with a brief overview of expressions. We then explained that you need to understand data types before you can build expressions and went on to discuss each of the major data types in some detail. We next showed you the CAST function and explained that you'll often use it to change the data type of a column or literal so that it's compatible with the type of expression you're trying to build. We then covered all the

ways that you can introduce a constant value—a literal— into your expressions. We then introduced you to the concept of using an expression to broaden or narrow the scope of information you retrieve from the database. We also explained that an expression is some form of operation involving numbers, character strings, or dates and times.

We continued our discussion of expressions and provided a concise overview of each type of expression. We showed you how to concatenate strings of characters and how to concatenate strings with other types of data by using the CAST function. We then showed you how to create mathematical expressions, and we explained how the order of precedence affects a given mathematical operation. We closed this discussion with a look at date and time expressions. After showing you how the SQL Standard handles dates and times, we revealed that most database systems provide their own methods of working with dates and times.

We then proceeded to the subject of using expressions in a SELECT statement, and we showed you how to incorporate expressions in the SELECT clause. We then showed you how to use both literal values and columns within an expression, as well as how to name the column that holds the result value of the expression. Before ending this discussion, we took a brief digression and introduced you to the value expression. We revealed that the SQL Standard uses this term to refer to a column reference, literal value, and expression collectively and that you can use a value expression in various clauses of an SQL statement. (More on this in later chapters, of course!)

We closed this chapter with a discussion on Nulls. You learned that a Null represents a missing or an unknown value. We showed you how to use a Null properly and explained that it can be quite useful under the right circumstances. But we also discussed how Nulls adversely affect mathematical operations. You now know that a mathematical operation involving a Null value returns a Null value. We also showed you how Nulls can make the information in a result set inaccurate.

In the next chapter, we'll discuss the idea of retrieving a very specific set of information. We'll then show you how to use a WHERE clause to filter the information retrieved by a SELECT statement.

The following section presents a number of requests that you can work out on your own.

Problems for You to Solve

Below, we show you the request statement and the name of the solution query in the sample databases. If you want some practice, you can work out the SQL for each request and then check your answer with the query we saved in the samples. Don't worry if your syntax doesn't exactly match the syntax of the queries we saved—as long as your result set is the same.

Sales Orders Database

1. *"What if we adjusted each product price by reducing it 5 percent?"*
   You can find the solution in CH05_Adjusted_Wholesale_Prices (90 rows).
2. *"Show me a list of orders made by each customer in descending date order."*
   (Hint: You might need to order by more than one column for the information to display properly.)
   You can find the solution in CH05_Orders_By_Customer_And_Date (944 rows).
3. *"Compile a complete list of vendor names and addresses in vendor name order."*
   You can find the solution in CH05_Vendor_Addresses (10 rows).

Entertainment Agency Database

1. *"Give me the names of all our customers by city."*
   (Hint: You'll have to use an ORDER BY clause on one of the columns.)
   You can find the solution in CH05_Customers_By_City (15 rows).
2. *"List all entertainers and their Web sites."*
   You can find the solution in CH05_Entertainer_Web_Sites (13 rows).
3. *"Show the date of each agent's first six-month performance review."*
   (Hint: You'll need to use date arithmetic to answer this request.)
   You can find the solution in CH05_First_Performance_Review (9 rows).

School Scheduling Database

1. *"Give me a list of staff members, and show them in descending order of salary."*
   You can find the solution in CH05_Staff_List_By_Salary (27 rows).
2. *"Can you give me a staff member phone list?"*
   You can find the solution in CH05_Staff_Member_Phone_List (27 rows).

3. *"List the names of all our students, and order them by the cities they live in."*
   You can find the solution in CH05_Students_By_City (18 rows).

Bowling League Database

1. *"Show next year's tournament date for each tournament location."*
   You can find the solution in CH05_Next_Years_Tourney_Dates (14 rows).
2. *"List the name and phone number for each member of the league."*
   You can find the solution in CH05_Phone_List (32 rows).
3. *"Give me a listing of each team's lineup."*
   (Hint: Base this query on the Bowlers table.)
   You can find the solution in CH05_Team_Lineups (32 rows).

6

Filtering Your Data

"I keep six honest-serving men
(They taught me all I knew.)
Their names are What and Why and When
and How and Where and Who."
—Rudyard Kipling
"I keep six honest-serving men"

Topics Covered in This Chapter

Refining What You See Using WHERE

Defining Search Conditions

Using Multiple Conditions

Nulls Revisited: A Cautionary Note

Expressing Conditions in Different Ways

Sample Statements

Summary

Problems for You to Solve

In the previous two chapters, we discussed the techniques you use to see all the information in a given table. We also discussed how to create and use expressions to broaden or narrow the scope of that information. In this chapter, we'll show you how to fine-tune what you retrieve by filtering the information using a WHERE clause.

Refining What You See Using WHERE

The type of SELECT statement we've worked with so far retrieves all the rows from a given table and uses them in the statement's result set. This is great if

you really do need to see all the information the table contains. But what if you want to find only the rows that apply to a specific person, a specific place, a particular numeric value, or a range of dates? These are not unusual requests. In fact, they are the impetus behind many of the questions you commonly pose to the database. You might, for example, have a need to ask the following types of questions.

"Who are our customers in Seattle?"

"Show me a current list of our Bellevue employees and their phone numbers."

"What kind of music classes do we currently offer?"

"Give me a list of classes that earn three credits."

"Which entertainers maintain a Web site?"

"Give me a list of engagements for the Caroline Coie Trio."

"Give me a list of customers who placed orders in May."

"Give me the names of our staff members who were hired on May 16, 1985."

"What is the current tournament schedule for Red Rooster Lanes?"

"Which bowlers are on team 5?"

In order to answer these questions, you'll have to expand your SQL vocabulary once again by adding another clause to our SELECT statement: the WHERE clause.

The WHERE Clause

You use a WHERE clause in a SELECT statement to filter the data the statement draws from a table. The WHERE clause contains a *search condition* that it uses as the filter. This search condition provides the mechanism needed to select only the rows you need or exclude the ones you don't want. Your database system applies the search condition to each row in the logical table defined by the FROM clause. Figure 6–1 shows the syntax of the SELECT statement with the WHERE clause.

A search condition contains one or more *predicates*, each of which is an expression that tests one or more value expressions and returns a true, false, or unknown answer. As you'll learn later, you can combine multiple predicates into a search condition using AND or OR Boolean operators. When the

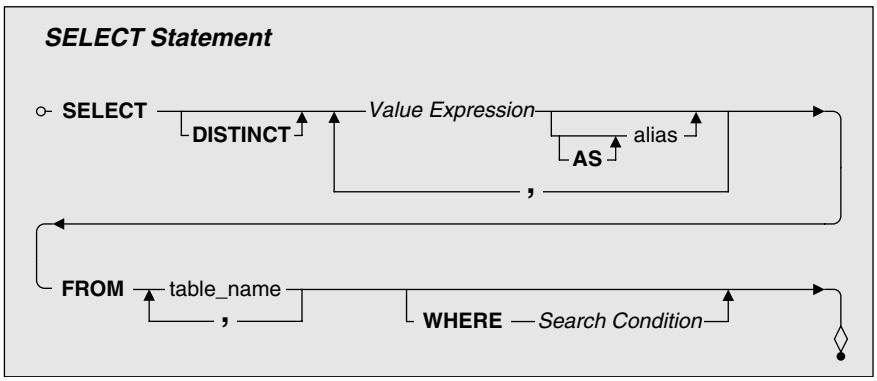


Figure 6–1 *The syntax diagram for a SELECT statement with a WHERE clause*

entire search condition evaluates to true for a particular row, you will see that row in the final result set. Note that when a search condition contains only one predicate, the terms *search condition* and *predicate* are synonymous.

Remember from Chapter 5, Getting More Than Simple Columns, that a value expression can contain column names, literal values, functions, or other value expressions. When you construct a predicate, you will typically include at least one value expression that refers to a column from the tables you specify in the FROM clause.

The simplest and perhaps most commonly used predicate compares one value expression (a column) to another (a literal). For example, if you want only the rows from the Customers table in which the value of the customer last name column is Smith, you write a predicate that compares the last name column to the literal value "Smith."

```
SQL         SELECT CustLastName
            FROM Customers
            WHERE CustLastName = 'Smith'
```

The predicate in the WHERE clause is equivalent to asking this question for each row in the Customers table: "Does the customer last name equal 'Smith'?" When the answer to this question is yes (true) for any given row in the Customers table, that row appears in the result set.

The SQL Standard defines eighteen predicates, but we'll cover the five basic ones in this chapter: Comparison, BETWEEN, IN, LIKE, and IS NULL.

COMPARISON	Use one of the six comparison operators (=, <>, <, >, <=, >=) to compare one value expression to another value expression.
BETWEEN (RANGE)	The BETWEEN predicate lets you test whether the value of a given value expression falls within a specified range of values. You specify the range using two value expressions separated by the AND keyword.
IN (MEMBERSHIP)	You can test whether the value of a given value expression matches an item in a given list of values using the IN predicate.
LIKE (PATTERN MATCH)	The LIKE predicate allows you to test whether a character string value expression matches a specified character string pattern.
IS NULL	Use the IS NULL predicate to determine whether a value expression evaluates to Null.

❖ **Note** Don't worry too much about the other thirteen predicates defined in the current SQL Standard. We could not find any commercial implementation of eleven of them. We'll cover the other two—Quantified and EXISTS—in Chapter 11, Subqueries.

Using a WHERE Clause

Before we explore each of the basic predicates in the SQL Standard, let's first take a look at another example of how to construct a simple WHERE clause. This time, we'll give you a detailed walkthrough of the steps to build your request.

❖ **Note** Throughout this chapter, we use the "Request/Translation/Clean Up/ SQL" technique introduced in Chapter 4, Creating a Simple Query.

Suppose you're making the following request to the database.

> *"What are the names of our customers who live in the state of Washington?"*

When composing a translation statement for this type of request, you must try to indicate the information you want to see in the result set as explicitly and clearly as possible. You'll expend more effort to rephrase a request than you've been accustomed to so far, but the results will be well worth the extra work. Here's how you translate this particular request.

Translation Select first name and last name from the customers table for those customers who live in Washington State

You'll clean up this statement in the usual fashion, but you'll also perform two extra tasks. First, look for any words or phrases that indicate or imply some type of restriction. Dead giveaways are the words "where," "who," and "for." Here are some examples of the types of phrases you're trying to identify.

"...who live in Bellevue."
"...for everyone whose ZIP Code is 98125."
"...who placed orders in May."
"...for suppliers in California."
"...who were hired on May 16, 1985."
"...where the area code is 425."
"...for Mike Hernandez."

When you find such a restriction, you're ready for the second task. Study the phrase, and try to determine which column is going to be tested, what value that column is going to be tested against, and how the column is going to be tested. The answers to these questions will help you formulate the search condition for your WHERE clause. Let's apply these questions to our translation statement.

Which column is going to be tested? **State**

What value is it going to be tested against? **'WA'**

How is the column going to be tested? **Using the "equal to" operator**

You need to be familiar with the structure of the table you're using to answer the request. If necessary, have a copy of the table structure handy before you begin to answer these questions.

❖ **Note** Sometimes the answers to these questions are evident, and other times the answers are implied. We'll show you how to make the distinction and decipher the correct answers as we work through other examples in this chapter.

After answering the questions, take them and create the appropriate condition. Next, cross out the original restriction, and replace it with the word WHERE and the search condition you just created. Here's how your Clean Up statement will look after you've completed this task.

Clean Up Select first name ~~and~~ last name from ~~the~~ customers ~~table for those customers who live in~~ where state ~~is equal to~~ = 'WA' ~~Washington State~~

Now you can turn this into a proper SELECT statement.

```
SQL         SELECT CustFirstName, CustLastName
            FROM Customers
            WHERE CustState = 'WA'
```

The result set of our completed SELECT statement will display only those customers who live in the state of Washington.

That's all there is to defining a WHERE clause. As we indicated at the beginning of this section, it's simply a matter of creating the appropriate search condition and placing it in the WHERE clause. The real work, however, is in defining the search conditions.

Defining Search Conditions

Now that you have an idea of how to create a simple WHERE clause, let's take a closer look at the five basic types of predicates you can define.

Comparison

The most common type of condition is one that uses a comparison predicate to compare two value expressions to each other. As you can see in Figure 6–2, you can define six different types of comparisons using the following comparison predicate operators.

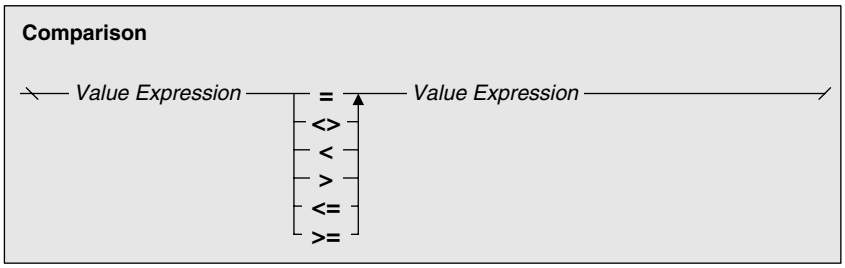


Figure 6–2 *The syntax diagram for the comparison condition*

= Equals	< Less Than	<= Less Than or Equal To
<> Not Equal To	> Greater Than	>= Greater Than or Equal To

Comparing String Values: A Caution

You can easily compare numeric or datetime data, but you must pay close attention when you compare character strings. For example, you might not get the results you expect when you compare two seemingly similar strings such as "Mike" and "MIKE." The determining factor for all character string comparisons is the collating sequence used by your database system. The collating sequence also determines how character strings are sorted and impacts how you use other comparison conditions as well.

Because many different vendors have implemented SQL on machines with different architectures and for many languages other than English, the SQL Standard does not define any default collating sequence for character string sorting or comparison. How characters are sorted from "lowest" to "highest" depends on the database software you are using and, in many cases, how the software was installed.

Many database systems use the ASCII collating sequence, which places numbers before letters and all uppercase letters before all lowercase letters. If your database supports the ASCII collating sequence, the characters are in the following sequence from lowest value to highest value.

. . . 0123456789 . . . ABC . . . XYZ . . . abc . . . xyz . . .

Some systems, however, offer a case-insensitive option. In these, for example, lowercase *a* is considered equal to uppercase *A*. When your database supports this option using ASCII as a base, characters are in the following sequence from lowest value to highest value.

. . . 0123456789 . . . {Aa}{Bb}{Cc} . . . {Xx}{Yy}{Zz} . . .

Note that the characters enclosed in braces ({}) are considered equal because no distinction is made between uppercase and lowercase. They sort alphabetically irrespective of the case.

Database systems running on IBM mainframe systems use the IBM-proprietary EBCDIC sequence. In a database system that uses EBCDIC, all lowercase letters come first, then all uppercase letters, and finally numbers. If your database supports EBCDIC, characters are in the following sequence from lowest value to highest value.

. . . abc . . . xyz . . . ABC . . . XYZ . . . 0123456789 . . .

To drive this point home, let's look at a set of sample column values to see how different collating sequences affect how your database system defines higher, lower, or equal values.

Here is a table of column values sorted using the ASCII character set, case sensitive (numbers first, then uppercase, and then lowercase).

Company Name
3rd Street Warehouse
5th Avenue Market
Al's Auto Shop
Ashby's Cleaners
Zebra Printing
Zercon Productions
allegheny & associates
anderson tree farm
zorn credit services
ztech consulting

Now, let's turn off case sensitivity so that lowercase letters and their uppercase equivalents are considered equal. The next table shows what happens.

Company Name
3rd Street Warehouse
5th Avenue Market
Al's Auto Shop
allegheny & associates
anderson tree farm
Ashby's Cleaners
Zebra Printing
Zercon Productions
zorn credit services
ztech consulting

Finally, let's see how these values are sorted on an IBM system using the EBCDIC collating sequence (lowercase letters, uppercase letters, and then numbers).

Company Name
allegheny & associates
anderson tree farm
zorn credit services
ztech consulting
Al's Auto Shop
Ashby's Cleaners
Zebra Printing
Zercon Productions
3rd Street Warehouse
5th Avenue Market

You can also encounter unexpected results when trying to compare two character strings of unequal length, such as "John" and "John " or "Mitch" and

"Mitchell." Fortunately, the SQL Standard clearly specifies how the database system must handle this. Before your database compares two character strings of unequal length, it must add the special *default pad character* to the right of the smaller string until it is the same length as the larger string. (The default pad character is a space in most database systems.) Your database then uses its collating sequence to determine whether the two strings are now equal to each other. As a result, "John" and "John " are equal (after the padding takes place) and "Mitch " and "Mitchell" are unequal.

❖ **Note** Some database systems differ from the SQL Standard in that they ignore trailing blanks rather than pad the shorter string with a default space. Therefore, "John" and "John " are considered equal in some systems, but for a different reason—because the trailing blanks in the second item are completely disregarded. Be sure to test your database system to determine how it handles this type of comparison and whether it returns the type of results you expect.

In summary, check your database system's documentation to determine how it collates uppercase letters, lowercase letters, and numbers.

Equality and Inequality

Although we've already seen a couple of examples, let's take another look at an equality comparison condition using the "equal to" operator.

Assume we're making this request to the database.

> *"Show me the first and last names of all the agents who were hired on March 14, 1977."*

Because we are going to search for a specific hire date, we can use an equality comparison condition with an "equal to" operator to retrieve the appropriate information. Now we'll run this through the translation process to define the appropriate SELECT statement.

Translation	Select first name and last name from the agents table for all agents hired on March 14, 1977
Clean Up	Select first name ~~and~~ last name from ~~the~~ agents ~~table for all agents hired on~~ where date hired = ~~March 14, 1977~~ '1977-03-14'

```
SQL         SELECT AgtFirstName, AgtLastName
            FROM Agents
            WHERE DateHired = '1977-03-14'
```

In this example, we tested the values of a specific column to determine whether any values matched a given date value. In essence, we executed an *inclusive* process—a given row in the Agents table will be included in the result set *only* if the current value of the DateHired column for that row matches the specified date. But what if you wanted to do the exact opposite and *exclude* certain rows from the result set? In that case, you would use a comparison condition with a "not equal to" operator.

Suppose you submit the following request.

> *"Give me a list of vendor names and phone numbers for all our vendors, with the exception of those here in Bellevue."*

You've probably already determined that you need to exclude those vendors based in Bellevue and that you'll use a "not equal to" condition for the task. The phrase "with the exception of" provides a clear indication that the "not equal to" condition is appropriate. Keep this in mind as you look at the translation process.

Translation	Select vendor name and phone number from the vendors table for all vendors except those based in 'Bellevue'
Clean Up	Select vendor name ~~and~~ phone number from ~~the~~ vendors ~~table for all vendors except those based in~~ where city <> 'Bellevue'

```
SQL         SELECT VendName, VendPhone
            FROM Vendors
            WHERE VendCity <> 'Bellevue'
```

❖ **Note** The SQL Standard uses the <> symbol for the "not equal to" operator. Several RDBMS programs provide alternate notations, such as != (supported by Microsoft SQL Server and Sybase) and ¬= (supported by IBM's DB2). Be sure to check your database system's documentation for the appropriate notation of this operator.

You've effectively excluded all vendors from Bellevue with this simple condition. Later in this chapter, we'll show you a different method for excluding rows from a result set.

Less Than and Greater Than

Often you want rows returned where a particular value in a column is smaller or larger than the comparison value. This type of comparison employs the "less than" (<), "less than or equal to" (<=), "greater than" (>), or "greater than or equal to" (>=) comparison operators. The type of data you compare determines the relationship between those values.

CHARACTER STRINGS	This comparison determines whether the value of the first value expression precedes (<) or follows (>) the value of the second value expression in your database system's collating sequence. For example, you can interpret *a* < *c* as "Does *a* precede *c*?" For details about collating sequences, see the previous section, Comparing String Values: A Caution.
NUMBERS	This comparison determines whether the value of the first value expression is smaller (<) or larger (>) than the value of the second value expression. For example, you can interpret *10* > *5* as "Is *10* larger than *5*?"
DATES/TIMES	This comparison determines whether the value of the first value expression is earlier (<) or later (>) than the value of the second value expression. For example, you can interpret *'2007-05-16'* < *'2007-12-15'* as "Is May 16, 2007, earlier than December 15, 2007?" Dates and times are evaluated in chronological order.

Let's take a look at how you might use these comparison predicates to answer a request.

> *"Are there any orders where the ship date was accidentally posted earlier than the order date?"*

You'll use a "less than" comparison operator in this instance because you want to determine whether any ship date was posted earlier than its respective order date. Here's how you translate this.

Translation	Select order number from the orders table where the ship date is earlier than the order date
Clean Up	Select order number from ~~the~~ orders ~~table~~ where ~~the~~ ship date ~~is earlier than the~~ < order date

SQL	`SELECT OrderNumber`
	`FROM Orders`
	`WHERE ShipDate < OrderDate`

The SELECT statement's result set will include only those rows from the Orders table where the search condition is true.

The next example requires a "greater than" comparison operator to retrieve the appropriate information.

"Are there any classes that earn more than four credits?"

Translation Select class ID from the classes table for all classes that earn more than four credits

Clean Up Select class ID from ~~the~~ classes ~~table for all classes that earn more than four~~ where credits > 4

SQL
```
SELECT ClassID
FROM Classes
WHERE Credits > 4
```

The result set generated by this SELECT statement includes only classes that earn five credits or more, such as Intermediate Algebra and Engineering Physics.

Now, let's take a look at some examples where you're interested not only in the values that might be greater than or less than but also equal to the comparison value.

"I need the names of everyone we've hired since January 1, 1989."

You use a "greater than or equal to" comparison for this because you want to retrieve all hire dates from January 1, 1989, to the present, *including* employees hired on that date. As you run through the translation process, be sure to identify all the columns you need for the SELECT clause.

Translation Select first name and last name as EmployeeName
from the employees table
for all employees hired since January 1, 1989

Clean Up Select first name ~~and~~ || ' ' || last name as EmployeeName
from ~~the~~ employees ~~table~~
~~for all employees hired since~~ where date hired >= ~~January 1, 1989~~ '1989-01-01'

SQL
```
SELECT FirstName || ' ' || LastName
    AS EmployeeName
FROM Employees
WHERE DateHired >= '1989-01-01'
```

Here's another request you might make to the database.

"Show me a list of products with a retail price of fifty dollars or less."

As you've probably deduced, you'll use a "less than or equal to" comparison for this request. This ensures that the SELECT statement's result set contains only those products that cost anywhere from one cent to exactly fifty dollars. Here's how you translate this request.

Translation	Select product name from the products table for all products with a retail price of fifty dollars or less
Clean Up	Select product name from ~~the~~ products ~~table for all products with a~~ where retail price ~~of~~ <= 50 ~~fifty dollars or less~~

SQL	`SELECT ProductName`
	`FROM Products`
	`WHERE RetailPrice <= 50`

The examples you've seen so far use only a single type of comparison. Later in this chapter, we'll show you how to combine comparisons using AND and OR.

Range

You can test the value of a value expression against a specific range of values with a range condition. Figure 6–3 shows the syntax for this condition.

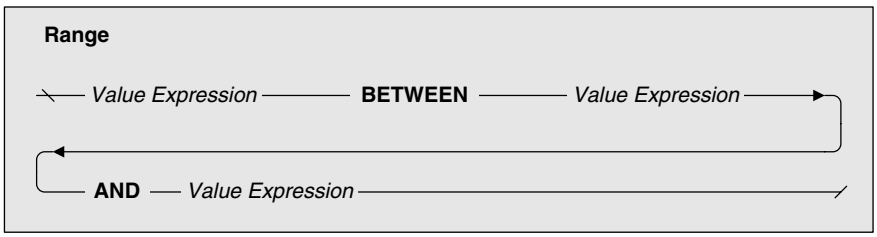


Figure 6–3 *The syntax diagram for the range condition*

The range condition tests the value of a given value expression against a range of values defined by two other value expressions. The BETWEEN . . . AND predicate defines the range by using the value of the second value expression as the start point and the value of the third value expression as the end point. Both the start point and end point are part of the range. A row is included in the result set only if the value of the first value expression falls within the specified range.

There's one "gotcha" about using BETWEEN . . . AND. The SQL Standard actually defines two types of BETWEEN comparisons: ASYMMETRIC and SYMMETRIC. The default, ASYMMETRIC, dictates that Value1 BETWEEN Value2 AND Value3 is the same as Value1 >= Value2 AND Value1 <= Value3. This means that Value2 must be less than or equal to Value3 for the predicate to work properly. For example, the SQL Standard states that

```
MyColumn BETWEEN 5 AND 10
```

should be processed as

```
MyColumn >= 5 AND MyColumn <= 10
```

So, putting the larger value first, as in

```
MyColumn BETWEEN 10 AND 5
```

is interpreted according to the SQL Standard as

```
MyColumn >=10 AND MyColumn <= 5
```

which can never be true! (The column value can't both be greater than or equal to 10 and at the same time less than or equal to 5.) However, some database systems allow Value2 to be greater than or equal to Value3—the equivalent of using the SYMMETRIC keyword in the SQL Standard. (We're not aware of any major implementation that yet supports the ASYMMETRIC and SYMMETRIC keywords.) Check your database system documentation for details.

Here are a couple of examples that illustrate how you use a range condition.

"Which staff members were hired in July 1986?"

The range condition is appropriate here because you want to retrieve the names of everyone who was hired within a specific set of dates, in this case, between July 1, 1986, and July 31, 1986. Let's now run this through the translation process and build the appropriate SELECT statement.

Translation	Select first name and last name from the staff table where the date hired is between July 1, 1986, and July 31, 1986
Clean Up	Select first name ~~and~~ last name from ~~the~~ staff ~~table~~ where ~~the~~ date hired ~~is~~ between ~~July 1, 1986~~ '1986-07-01' and ~~July 31, 1986~~ '1986-07-31'

```
SQL         SELECT FirstName, LastName
            FROM Staff
            WHERE DateHired
               BETWEEN '1986-07-01' AND '1986-07-31'
```

Notice that we stated the range of dates more explicitly in the translation statement than in the request. Use this technique to translate the request as clearly as possible and thus define the appropriate SELECT statement.

You can also use a range condition on character string data quite effectively, as shown in this example.

"Give me a list of students—along with their phone numbers—whose last names begin with the letter B."

Translation Select last name, first name, and phone number
from the students table
for all students whose last name begins with the letter 'B'

Clean Up Select last name, first name, ~~and~~ phone number
from ~~the~~ students ~~table~~
~~for all students whose name begins with the letter 'B'~~
where last name between 'B' and 'Bz'

```
SQL         SELECT StudLastName, StudFirstName, StudPhoneNumber
            FROM Students
            WHERE StudLastName BETWEEN 'B' AND 'Bz'
```

When creating a range for character string data, think carefully about the values you want to include. For example, here are three possible ways you might have indicated the start and end points for the required range in this request. The results are quite different!

BETWEEN 'A' AND 'C' We know that many of you would not have indicated 'A' as the start point because you know the range would then include everyone whose name begins with that letter. However, this is a fairly typical mistake.

BETWEEN 'B' AND 'C' Indicating the start and end points in this manner probably returns the desired results for our example. However, you might get unexpected results based on the character data you're trying to compare. Remember that the BETWEEN operator *includes* the start and end points in the range. Consequently, a student whose last name is only the letter 'C' will be included in the result set.

BETWEEN 'B' AND 'Bz'	This is the clearest and most explicit method of indicating the start and end points—in most cases, it will return the desired results. In the end, you must understand your data in order to define the correct range.

One more thing before we leave BETWEEN. Notice that the diagram in Figure 6–3 says that you can use a *value expression* not only for the two values in the BETWEEN clause but also for the first value (see page 164). As we've explained, a value expression can be as simple as a column name or a simple literal or as complex as a character, mathematical, or datetime expression. When you have a table that has two columns that define a range of values (for example, StartDate and EndDate in the Engagements table in the Entertainment Agency sample database), you can also use BETWEEN to search for rows that contain a value BETWEEN the values in the two columns. Here's an example.

"Show me all engagements that are scheduled to occur on October 10, 2007."

Translation	Select engagement number, start date, and end date from the engagements table for engagements where October 10, 2007, is between the start date and the end date
Clean Up	Select engagement number, start date, ~~and~~ end date from ~~the~~ engagements ~~table~~ ~~for engagements~~ where ~~October 10, 2007 is~~ '2007-10-10' between ~~the~~ start date and ~~the~~ end date

SQL	`SELECT EngagementNumber, StartDate, EndDate`
	`FROM Engagements`
	`WHERE '2007-10-10' BETWEEN StartDate AND EndDate`

So far, we've shown you how to narrow the scope of your request using a broad range of values and a more specific range of values. Now, let's take a look at how you can refine your requests even further by using an explicit list of values.

Set Membership

You'll use the membership condition to test the value of a value expression against a list of explicitly defined values. As you can see in Figure 6–4 (on page 168), the membership condition uses the IN predicate to determine whether the value of the first value expression matches any value within a parenthetical list of values defined by one or more value expressions.

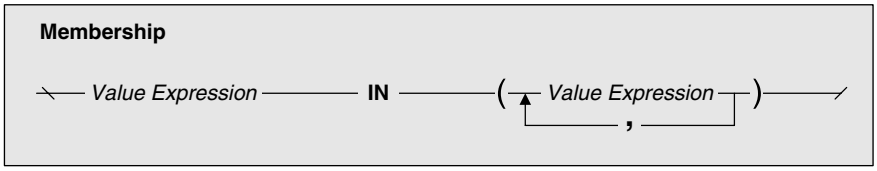


Figure 6–4 *The syntax diagram for the membership condition*

Although theoretically you can include an almost limitless number of value expressions in the list, it makes more sense to use only a few. You already have two conditions at your disposal that you can use to indicate broader ranges of values. You can use the membership condition most effectively when you define a finite list of values, as you'll see in the following examples.

Here's a request you might make to the database.

> *"I need to know which bowling lanes sponsored tournaments for the following 2007 dates: September 18, October 9, and November 6."*

This type of request lends itself to a membership condition because it focuses on searching for a specific set of values. If the request were not so explicit, you would most likely use a range condition instead. Here's how to translate this request.

Translation	Select tourney location from the tournaments table where the tourney date is in this list of dates: September 18, 2007; October 9, 2007; November 6, 2007
Clean Up	Select tourney location from ~~the~~ tournaments ~~table~~ where ~~the~~ tourney date ~~is~~ in ~~this list of dates:~~ (~~September 18, 2007;~~ '2007-09-18',~~October 9 2007;~~ '2007-10-09', ~~November 6, 2007~~ '2007-11-06')

SQL	`SELECT TourneyLocation`
	`FROM Tournaments`
	`WHERE TourneyDate`
	`  IN ('2007-09-18', '2007-10-09',`
	`  '2007-11-06')`

Here's another request that requires a membership condition for its answer.

> *"Which entertainers do we represent in Seattle, Redmond, and Bothell?"*

Translation	Select stage name from the entertainers table for all entertainers based in 'Seattle', 'Redmond', or 'Bothell'
Clean Up	Select stage name from ~~the~~ entertainers ~~table for all entertainers based~~ where city in ('Seattle','Redmond', ~~or~~ 'Bothell')

```
SQL         SELECT EntStageName
            FROM Entertainers
            WHERE EntCity
               IN ('Seattle', 'Redmond', 'Bothell')
```

You might have noticed that we used the word "or" in the translation statement's list of cities instead of "and" as it appears in the original request. The reason and logic for this is simple: There is only one entry in the EntCity column for a given entertainer. A given row can't contain Seattle *and* Redmond *and* Bothell all at the same time, but a single row could contain Seattle *or* Redmond *or* Bothell. This might seem a trivial point, but using the proper words and phrases helps to clarify your Translation and Clean Up statements and ensures that you define the most appropriate SELECT statement for your request. You'll see that this small point becomes even more important later in the chapter when you begin using multiple conditions.

All the conditions you've learned so far use complete values as their criteria. Now we'll take a look at a condition that allows you to use partial values as a criterion.

Pattern Match

The pattern match condition is useful when you need to find values that are similar to a given pattern string or when you have only a partial piece of information to use as a search criterion. Figure 6–5 shows the syntax for this type of condition.

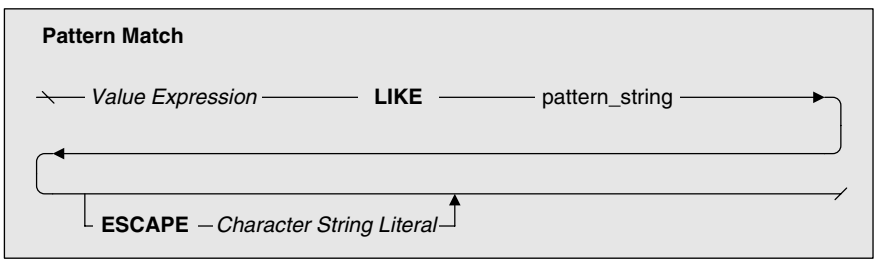


Figure 6–5 *The syntax diagram for the pattern match condition*

This condition takes the value of a value expression and uses the LIKE predicate to test whether the value matches a defined pattern string. A pattern string can consist of any logical combination of regular string characters and two special wildcard characters: the percent sign (%) and the underscore (_). The percent sign represents zero or more arbitrary regular characters, and the

underscore represents a single arbitrary regular character. The manner in which you define the pattern string determines which values are retrieved from the value expression. Table 6–1 shows samples of the different types of pattern strings you can define.

❖ **Note** One of the most popular database systems, Microsoft Office Access, uses an asterisk (*) instead of the percent sign (%) and a question mark (?) instead of an underscore (_). Access also supports using an octothorp (#) to search for numeric characters in specific positions. If you're using Microsoft Access, substitute these characters in your pattern strings for the LIKE predicate.

Table 6–1 *Samples of Defined Pattern Strings*

Pattern String	Criterion Processed	Sample Return Values
`'Sha%'`	Character string can be any length but must begin with "Sha"	**Sha**nnon, **Sha**ron, **Sha**wn
`'%son'`	Character string can be any length but must end with "son"	Ben**son**, John**son**, Morri**son**
`'%han%'`	Character string can be any length but must contain "han"	Buc**han**an, **han**del, Jo**han**sen, Nat**han**son
`'Ro_'`	Character string can be only three characters in length and must have "Ro" as the first and second letters	**Ro**b, **Ro**n, **Ro**y
`'_im'`	Character string can be only three characters in length and must have "im" as the second and third letters	J**im**, K**im**, T**im**
`'_ar_'`	Character string can be only four characters in length and must have "ar" as the second and third letters	B**ar**t, G**ar**y, M**ar**k
`'_at%'`	Character string can be any length but must have "at" as the second and third letters	G**at**es, M**at**thews, P**at**terson
`'%ac_'`	Character string can be any length but must have "ac" as the second and third letters from the end of the string	Apod**ac**a, Tr**ac**y, Wall**ac**e

Let's take a look at how you can use a pattern match condition by considering the following request.

> *"Give me a list of customers whose last names begin with 'Mar'."*

Requests such as this one typically use phrases that indicate the need for a pattern match condition. Here are a few examples of the types of phrases you're likely to encounter.

"...begin with 'Her'."
"...start with 'Ba'."
"...include the word 'Park'."
"...contain the letters 'han'."
"...have 'ave' in the middle of it."
"...with 'son' at the end."
"...ending in 'ez'."

❖ **Caution** In most database systems, string comparison is case sensitive. Several major database systems allow system administrators to specify an option to use either case-sensitive or case-insensitive comparison when they install database servers. If your database system is case sensitive, `LIKE '%chi%'` will find "roast chicken," but it won't find "Chicken a la King" because the lowercase 'c' in the pattern string is not equal to the uppercase 'C' in the column. Check your database documentation to find out whether you need to deal with the difference between upper- and lowercase letters.

As you can see, it can be relatively easy to determine the type of pattern string you need for a request. After you know the type of pattern you need to create, you can continue with the translation process.

Translation	Select last name and first name from the customers table where the last name begins with 'Mar'
Clean Up	Select last name ~~and~~ first name from ~~the~~ customers ~~table~~ where ~~the~~ last name ~~begins with~~ like 'Mar%'

SQL	`SELECT CustLastName, CustFirstName`
	`FROM Customers`
	`WHERE CustLastName LIKE 'Mar%'`

The result set for this SELECT statement includes names such as Marks, Marshall, Martinez, and Marx because we were only concerned with matching the first three letters of the last name.

Here's how you might answer another request using a pattern match condition.

"Show me a list of vendor names where the word 'Forest' appears in the street address."

Translation: Select vendor name from the vendors table where the street address contains the word 'Forest'

Clean Up: Select vendor name from ~~the~~ vendors ~~table~~ where ~~the~~ street address ~~contains the word~~ like '%Forest%'

SQL:
```
SELECT VendName
FROM Vendors
WHERE VendStreetAddress LIKE '%Forest%'
```

In this case, a row from the Vendors table is included in the result set only if the street address contains a street name such as Forest Park Place, Forest Ridge Avenue, Evergreen Forest Drive, or Black Forest Road.

Although you can search for any pattern string using the appropriate wildcard characters, you'll run into a problem if the values you want to retrieve include a percent sign or an underscore character. For example, you will have a problem trying to retrieve the value MX_445 because it contains an underscore character. You can circumvent this potential dilemma by using the ESCAPE option of the LIKE predicate, as shown in Figure 6–5.

The ESCAPE option allows you to designate a *single* character string literal—known as an *escape character*—to indicate how the database system should interpret a percent sign or underscore character within a pattern string. Place the escape character after the ESCAPE keyword and enclose it within single quotes, as you would any character string literal. When the escape character precedes a wildcard character in a pattern string, the database system interprets that wildcard character *literally* within the pattern string.

Here's an example of how you might use the ESCAPE option.

"Show me a list of products that have product codes beginning with 'G_00' and ending in a single number or letter."

Translation: Select product name and product code from the products table where the product code begins with 'G_00' and ends in a single number or letter

Clean Up	Select product ~~name and~~ product code from ~~the~~ products ~~table~~ where ~~the~~ product code ~~begins with~~ like 'G_00_' ~~and ends in a single number or letter~~

SQL	`SELECT ProductName, ProductCode`
	`FROM Products`
	`WHERE ProductCode LIKE 'G\_00_' ESCAPE '\'`

It's evident that you need to use the ESCAPE option to help answer this request—otherwise, the database system interprets the underscore character in the pattern string as a wildcard character. Note that we included the escape character in the Clean Up statement. You should do so in your Clean Up statements as well because it ensures that you remember to use the ESCAPE option when you define your SELECT statement.

This SELECT statement will retrieve product codes such as G_002 and G_00X. Because we want to search for one of the two characters that are defined in the standard as a wildcard, we *must* include the ESCAPE clause. If we ask for `LIKE 'G_00_'`, the database system will return rows where the product code has a 'G' for the first letter, *any* character in the second position (because of the wildcard character), zeros in the third and fourth positions, and any character in the fifth position. When we define "\" as the escape character, the database system ignores the escape character but interprets the first underscore character literally, not as a wildcard. Because we did not use the escape character just before the second underscore, the database system interprets the second underscore as a true wildcard character.

Keep in mind that the character you use as an escape character should not be part of the values you're trying to retrieve. It doesn't make sense to use & as an escape character if you're searching for values such as Martin & Lewis, Smith & Kearns, or Hernandez & Viescas. Also remember that the escape character affects only the wildcard character that immediately follows it. However, you can use as many escape characters in your pattern string as are appropriate.

Null

Now that you've learned how to search for complete values and partial values, let's discuss searching for *unknown* values. You learned in Chapter 5 that a Null *does not* represent a zero, a character string of one or more blank spaces, or a zero-length character string (a character string that has no characters in it) because each of these items can be meaningful in a variety of cir-

cumstances. You also learned that a Null *does* represent a missing or unknown value. To retrieve Null values from a value expression, you use the *Null condition* shown in Figure 6–6.

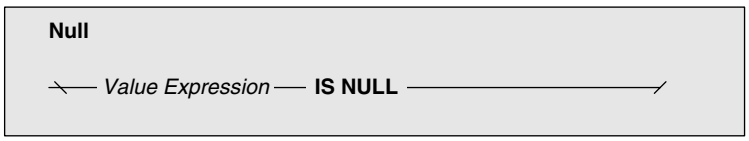


Figure 6–6 *The syntax diagram for the Null condition*

This condition takes the value of the value expression and determines whether it is Null using the IS NULL predicate. It's quite a straightforward operation. Let's take a look at how you might use this condition on the following examples.

"Give me a list of customers who didn't specify what county they live in."

Translation: Select first name and last name as Customer from the customers table where the county name is unspecified

Clean Up: Select first name || '' || ~~and~~ last name as Customer from ~~the~~ customers ~~table~~ where ~~the~~ county name is null ~~unspecified~~

SQL:

```
SELECT CustFirstName || ' ' || CustLastName
   AS Customer
FROM Customers
WHERE CustCounty IS NULL
```

The only customers who appear in the result set for this SELECT statement are those who didn't know or couldn't remember what county they live in, or those folks who live in Washington, D.C. (Washington, by the way, is the only city in the entire United States that isn't situated within a county.)

Here's another request you might make to the database.

"Which engagements do not yet have a contract price?"

Translation: Select engagement number and contract price from the engagements table for any engagement that does not have a contract price

Clean Up: Select engagement number ~~and~~ contract price from ~~the~~ engagements ~~table for any engagement that does not have a~~ where contract price is null

```
SQL         SELECT EngagementNumber, ContractPrice
            FROM Engagements
            WHERE ContractPrice IS NULL
```

On the surface, this seems like a straightforward request—you'll just search for any engagement that has 0 as the contract price. But looks can be deceiving, and they can lull you into making incorrect assumptions. If the entertainment agency in this example uses 0 as the contract price for any promotional engagement, then zero is a valid, meaningful value. Therefore, any contract price that is yet to be determined or negotiated is indeed (or should be) Null.

This example illustrates the fact that you do need to understand your data in order to make meaningful, accurate requests to the database. If you execute a SELECT statement and then think that the information you see in a result set is erroneous, don't panic. Your first impulse will probably be to rewrite the entire SELECT statement because you believe you've made some disastrous mistake in the syntax. Before you do anything drastic, review the data you're working with, and make certain you have a clear idea of how it's being used. After you have a better understanding of the data, you'll often find that you need to make only minor changes to your SELECT statement in order for it to retrieve the proper information.

❖ **Note** You must use the Null condition to search for Null values within a value expression. A condition such as `<ValueExpression> = Null` is invalid because the value of the value expression cannot be compared to something that is, by definition, unknown.

Excluding Rows with NOT

Up to this point, we've shown you how to *include* specific rows in a result set. Let's now take a look at how you *exclude* rows from a result set by using the NOT operator. We've already shown you one simple way to exclude rows from a result set by using an equality comparison condition with a "not equal to" operator. You can also exclude rows with other types of conditions by using the NOT operator. As you can see in Figure 6–7 (on page 176), this operator is an optional component of the BETWEEN, IN, LIKE, and IS NULL predicates. A SELECT statement will disregard any rows that meet the condition expressed by any of these predicates when you include the NOT operator. The rows that will be in the result set instead are those that *did not meet* the condition.

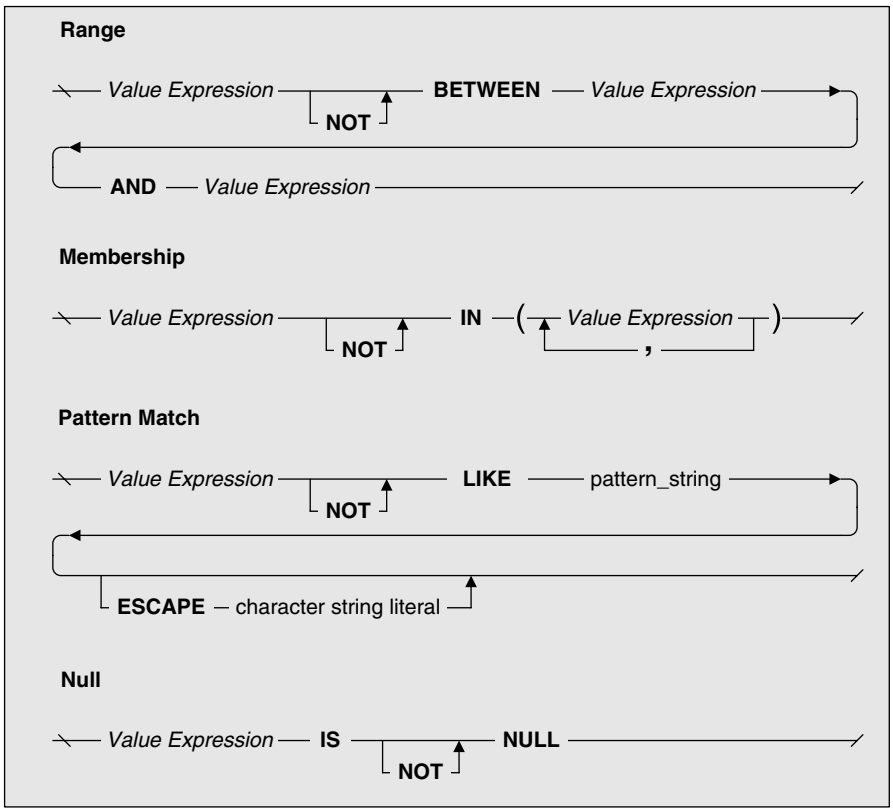


Figure 6–7 *The syntax diagram for the NOT operator*

The following examples illustrate how you can use NOT as part of a search condition.

> *"Show me a list of all the orders we've taken, except for those posted in July."*

A request such as this requires you to define a SELECT statement that excludes rows meeting a specific criterion and commonly contains phrases that indicate the need for a NOT operator as part of the search condition. The types of phrases you'll encounter are similar to those listed here.

> *"... that don't begin with 'Her'."*
> *"... that aren't in the Administrative or Personnel departments."*
> *"... who have a fax number."*
> *"... who were hired before June 1 or after August 31."*

You have to perform a bit of deductive work sometimes in order to translate a phrase properly. Some phrases, such as the third phrase listed above, do not explicitly indicate the need for a NOT operator. In this case, the requirement is implied because you want to *exclude* everyone who *does not* have a fax number. As you begin to work with requests that contain these types of phrases, you'll often find that you need to analyze them carefully and possibly rewrite them in order to determine whether you need to exclude certain rows from the result set. There's no easy rule of thumb we can give you here, but with a little patience and practice it will become easier for you to determine whether you need a NOT operator for a specific request.

After you've determined whether you need to exclude any information from the result set, you can continue with the translation process.

"Show me a list of all the orders we've taken, except for those posted in October."

Translation	Select order ID and order date from the orders table where the order date does not fall between October 1, 2007, and October 31, 2007
Clean Up	Select order ID ~~and~~ order date from ~~the~~ orders ~~table~~ where ~~the~~ order date ~~does~~ not ~~fall~~ between ~~October 1, 2007,~~ '2007-10-01' and ~~October 31, 2007~~ '2007-10-31'

SQL	`SELECT OrderID, OrderDate`
	`FROM Orders`
	`WHERE OrderDate NOT BETWEEN '2007-10-01'`
	`  AND '2007-10-31'`

This SELECT statement produces a result set that will *not* contain any orders posted between October 1, 2007, and October 31, 2007. It will, however, contain every other order in the Orders table. You can further restrict the rows sent to the result set to only those orders taken in 2007 by using multiple conditions, which is an issue we'll cover in the next section.

Now let's assume you're working with the following request.

"I need the identification numbers of all faculty members who are not professors or associate professors."

Translation	Select staff ID and title from the faculty table where the title is not 'professor' or 'associate professor'
Clean Up	Select staff ID ~~and~~ title from ~~the~~ faculty ~~table~~ where ~~the~~ title ~~is~~ not in ('professor', ~~or~~ 'associate professor')

```
SQL         SELECT StaffID, Title
            FROM Faculty
            WHERE Title
            NOT IN ('Professor', 'Associate Professor')
```

In this case, you need to exclude any staff member whose title is one of those specified within the request, so you use a membership condition with a NOT operator to send the correct rows to the result set.

Excluding rows from a result set becomes a relatively straightforward process after you get accustomed to analyzing and rephrasing your requests as the situation dictates. The real key, as you've seen so far, is being able to determine the type of condition you need to answer a given request.

Using Multiple Conditions

The requests we've worked with up to this point have been simple and have required only a single condition to supply the answer. Now we'll look at how you can answer complex requests using multiple conditions. Let's begin by considering the following request.

> *"Give me the first and last names of customers who live in Seattle and whose last names start with the letter 'H'."*

Based on the knowledge you've gained thus far, you can ascertain that this request requires an equality comparison condition and a pattern match condition to supply an answer. You've identified the conditions you need, but how do you combine them into one search condition? The answer lies in the way the SQL Standard defines the syntax for a search condition, as shown in Figure 6–8.

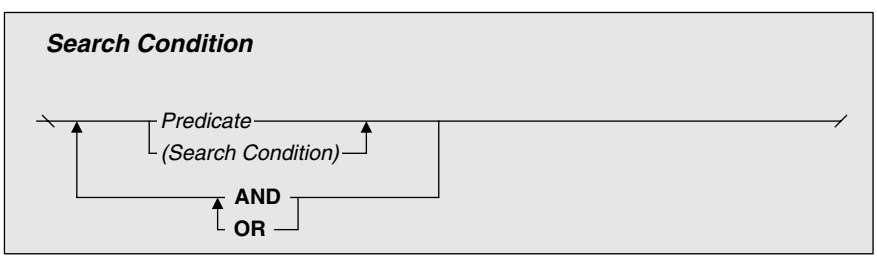


Figure 6–8 *The syntax diagram for the search condition*

Introducing AND and OR

You can combine two or more conditions by using the AND and OR operators, and the complete set of conditions you've combined to answer a given request constitutes a single search condition. As Figure 6–8 shows, you can also combine a complete search condition with other conditions by enclosing the search condition in parentheses. All this allows you to create very complex WHERE clauses that precisely control which rows are selected to be included in a result set.

Using AND

The first way you can combine two or more conditions is by using the AND operator. You use this operator when *all* the conditions you combine must be met in order for a row to be included in a result set. Let's use the sample request we made at the beginning of this section as an example and apply this operator during the translation process.

"Give me the first and last names of customers who live in Seattle and whose last names start with the letter 'H'."

Translation	Select first name and last name from the customers table where the city is 'Seattle' and the last name begins with 'H'
Clean Up	Select first name ~~and~~ last name from ~~the~~ customers ~~table~~ where ~~the~~ city ~~is~~ = 'Seattle' and ~~the~~ last name ~~begins with~~ like 'H%'

SQL

```
SELECT CustFirstName, CustLastName
FROM Customers
WHERE CustCity = 'Seattle'
   AND CustLastName LIKE 'H%'
```

You've accounted for both the equality comparison condition and the pattern match condition required by the request, and you've ensured that they must both be met by using the AND operator. Any row that fails to meet either condition will be excluded from the result set.

You can chain any number of conditions you need to answer the request at hand. Just keep in mind that *all* the conditions you've combined with ANDs *must* be met in order for a row to be included in the result set. Remember that the entire search condition must evaluate to true for a row to appear in the result set. Figure 6–9 (on page 180) shows the result when you combine two predicate expressions using the AND operator. If *either* of the expressions evaluates to false, then the row is not selected.

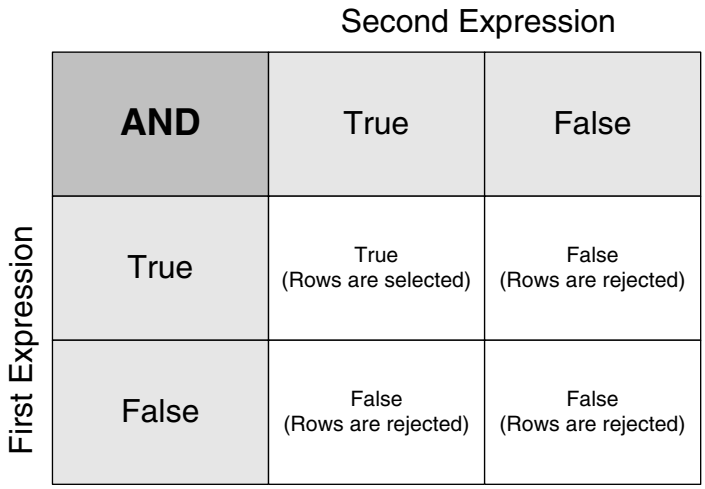


Figure 6–9 *The result of combining two predicate expressions with the AND operator*

Using OR

The second way to combine two or more conditions is by using the OR operator. You use this operator when *either* of the conditions you combine can be met in order for a row to be included in a result set. Here's an example of how you might use an OR operator in a search condition.

"I need the name, city, and state of every staff member who lives in Seattle or is from the state of Oregon."

Translation	Select first name, last name, city, and state from the staff table where the city is 'Seattle' or the state is 'OR'
Clean Up	Select first name, last name, city, ~~and~~ state from ~~the~~ staff ~~table~~ where ~~the~~ city ~~is~~ = 'Seattle' or ~~the~~ state ~~is~~ = 'OR'
SQL	`SELECT StfFirstName, StfLastName, StfCity, StfState FROM Staff WHERE StfCity = 'Seattle' OR StfState = 'OR'`

In this case, you've accounted for both of the equality comparison conditions you need to answer this request, and you've ensured that *only one* of the conditions has to be met by using the OR operator. As long as a row fulfills either

condition, it will be included in the result set. To help clarify the matter, Figure 6–10 shows the result of combining two predicate expressions with an OR operator.

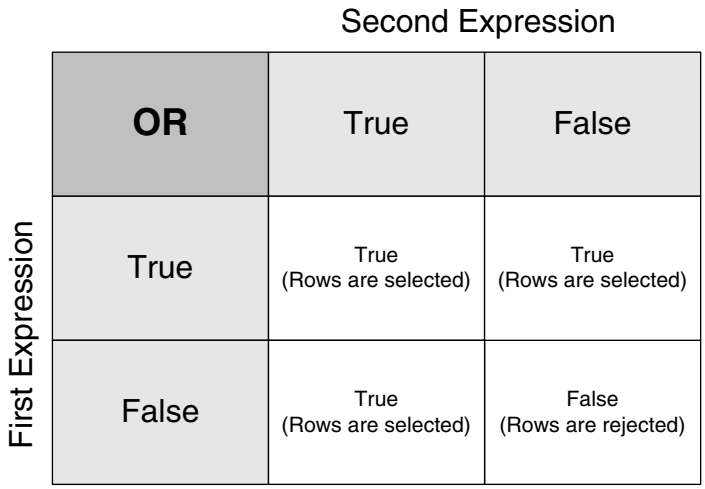


Figure 6–10 *The result of combining two predicate expressions with the OR operator*

Determining whether to use an AND operator to combine conditions is relatively easy and straightforward. However, determining whether to use an OR operator can be tricky sometimes. For example, consider the following request.

> *"Show me a list of vendor names and phone numbers for all vendors based in Washington and California."*

Your first impulse might be to use an AND operator because the condition seems obvious—you want vendors in Washington *and* California. Unfortunately, you would be wrong. If you think about it, a vendor will be based in *either* Washington *or* California because you can enter only one state value in the state column for that vendor. The actual condition is much clearer now, isn't it? As we mentioned earlier in the chapter, you must get into the habit of studying and analyzing your requests as they become more complex. Try to look for implied conditions as best as you can.

Let's continue and run this request through the translation process.

"Show me a list of vendor names and phone numbers for all vendors based in Washington and California."

Translation	Select name, phone number, and state from the vendors table where the state is 'WA' or 'CA'
Clean Up	Select name, phone number, ~~and~~ state from ~~the~~ vendors ~~table~~ where ~~the~~ state ~~is~~ = 'WA' or state = 'CA'

SQL	`SELECT VendName, VendPhoneNumber, VendState`
	`FROM Vendors`
	`WHERE VendState = 'WA' OR VendState = 'CA'`

You've accounted for both equality comparison conditions and ensured that either one must be met by using the OR operator. Note, however, that "state" appears in the search condition of the Clean Up and SQL statements twice. This is necessary because each comparison condition follows the same syntax:

```
Value Expression <comparison operator> Value Expression
```

Remember that you cannot omit any clause, keyword, or defined term from the syntax unless it is explicitly defined as an optional item. Thus, a condition such as `WHERE VendState = 'WA' OR 'CA'` is completely invalid. You might ask why this is so. We'll explain more about the sequence in which expression operators get evaluated—the order of precedence—later.

In this case, your database system evaluates the expression in strict left-to-right sequence. So, `VendState = 'WA'` will be evaluated first. For any given row, the result will be true if the state is Washington, and false if it is not. Next, this true or false result gets "ORed" with the literal value `'CA'`—which is not a true or false value! Your database system might return an error at this point (`'CA'`—a character string literal—is an invalid data type for the OR operator), or it might return only the rows where the state is Washington.

Always make certain that your conditions are completely and correctly defined. Otherwise, the search condition for your SELECT statement will fail.

❖ **Note** We used this example to illustrate a common trap you'll encounter when you use the OR operator. However, if you thought you could use a membership condition such as `WHERE VendState IN ('WA', 'CA')` to answer this request, you are absolutely correct. In some instances, you'll find that there's more than one way to express a condition.

Using AND and OR Together

You can use both AND and OR to answer particularly tricky requests. For example, you can answer the following type of request by using both operators.

> *"I need to see the names of staff members who have a 425 area code and a phone number that begins with 555, along with anyone who was hired between October 1 and December 31 of 2007."*

It should be easy for you to decide what types of conditions you need for this request by now. You've probably already determined that you need three conditions to answer this request: an equality comparison condition to find the area code, a pattern match condition to find the phone numbers, and a range condition to find those staff members hired between October 1 and December 31. All you have to do now is determine how you're going to combine the conditions.

You need to combine the comparison and pattern match conditions with an AND operator because they identify the phone numbers you're searching for and because both conditions must be met in order for a row to be included in the result set. You then treat this combination of conditions as a single unit and combine it with the range condition using an OR operator. Now a row will be included in the result set as long as it meets *either* the combined condition or the range condition.

Here's the request again and the translation.

> *"I need to see the names of staff members who have a 425 area code and a phone number that begins with 555, along with anyone who was hired between October 1 and December 31 of 2007."*

Translation	Select first name, last name, area code, phone number, and date hired from the staff table where the area code is 425 and the phone number begins with 555 or the date hired falls between October 1, 2007, and December 31, 2007
Clean Up	Select first name, last name, area code, phone number, ~~and~~ date hired from ~~the~~ staff ~~table~~ where ~~the~~ area code ~~is~~ = '425' and ~~the~~ phone number ~~begins with~~ like '555%' or ~~the~~ date hired ~~falls~~ between ~~October 1, 2007,~~ '2007-10-01' and ~~December 31, 2007~~ '2007-12-31'
SQL	`SELECT StfFirstName, StfLastName, StfAreaCode, StfPhoneNumber, DateHired FROM Staff`

```
WHERE (StfAreaCode = '425'
   AND StfPhoneNumber LIKE '555%')
OR DateHired
   BETWEEN '2007-10-01' AND '2007-12-31'
```

The previous example clearly demonstrates a situation where you can use a search condition within a search condition. Before you translated the request, we said that you needed to combine the comparison and pattern match conditions with an AND operator and then treat them as a single unit. When you treat a combined set of conditions as a single unit, by definition it becomes a search condition, and you must enclose it in parentheses, exactly as we did in the example.

Here's another example using AND and OR.

"I need the name and title of every professor or associate professor who was hired on May 16, 1989."

Translation: Select first name, last name, title, and date hired from the staff table where the title is 'professor' or 'associate professor' and the date hired is May 16, 1989

Clean Up: Select first name, last name, title, ~~and~~ date hired from ~~the~~ staff ~~table~~ where ~~the~~ title ~~is~~ = 'professor' or title = 'associate professor' and ~~the~~ date hired ~~is~~ = ~~May 16, 1989~~ '1989-05-16'

SQL:

```
SELECT StfFirstName, StfLastName, Title, DateHired
FROM Staff
WHERE (Title = 'Professor' OR Title =
   'Associate Professor') AND DateHired =
   '1989-05-16'
```

You've probably guessed that the two conditions combined with the OR operator are being treated as a single search condition. This example merely reinforces the fact that you can define a search condition with either the AND or the OR operators. But once again, the key is making certain that you enclose the search condition within parentheses.

Excluding Rows: Take Two

If you're feeling a bit of déjà vu, don't worry—we did discuss this already. Well, at least to some extent. You learned earlier in this chapter that the NOT operator is an option of the BETWEEN, IN, LIKE, and IS NULL predicates. But as Figure 6–11 illustrates, NOT is also an option as the first keyword of a

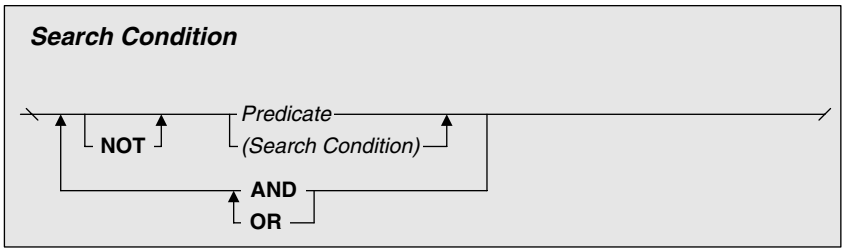


Figure 6–11 *Including the NOT operator in a search condition*

search condition, and it allows you to exclude rows from a result set just as you can by using NOT within a predicate. You use this particular NOT operator *before* a single condition (predicate) or an embedded search condition. Once again, you can express the same condition in various ways.

Let's assume you're posing the following request to the database.

> *"Show me the location and date of any tournament not being held at Bolero Lanes, Imperial Lanes, or Thunderbird Lanes."*

You've probably already determined that you'll use a membership condition to answer this request. Now you just need to determine how you'll define it. One approach you can take is using the NOT operator within the predicate.

```
WHERE TourneyLocation NOT IN ('Bolero Lanes',
   'Imperial Lanes', 'Thunderbird Lanes')
```

Another approach you might consider is using the NOT operator as the first keyword before the search condition.

```
WHERE NOT TourneyLocation IN ('Bolero Lanes',
   'Imperial Lanes', 'Thunderbird Lanes')
```

Either condition will exclude tournaments held at Bolero Lanes, Imperial Lanes, and Thunderbird Lanes from the result set. However, one advantage of using NOT before a search condition is that you can apply it to a comparison condition. (Remember that the syntax for a comparison condition does not include NOT as an optional operator.) But now you can use a comparison condition to exclude rows from a result set. The following example shows how you might use this type of condition.

"Show me the bowlers who live outside of Bellevue."

Translation	Select first name, last name, and city from the bowlers table where the city is not 'Bellevue'
Clean Up	Select first name, last name, ~~and~~ city from ~~the~~ bowlers ~~table~~ where ~~the~~ city ~~is~~ not = 'Bellevue'

SQL	`SELECT BowlerFirstName, BowlerLastName, BowlerCity`
	`FROM Bowlers`
	`WHERE NOT BowlerCity = 'Bellevue'`

Yes, we know that you could have expressed this condition as `WHERE BowlerCity <> 'Bellevue'`. This example simply emphasizes that you can express a condition in various ways.

Now that you've learned how to use a NOT operator within a single condition and a complete search condition, be aware of a problem that can occur when you define a search condition with two NOT operators that will *include* rows instead of *excluding* them. Here's an example.

"Which staff members are not teachers or teacher's aides?"

Translation	Select first name, last name, and title from the staff table where the title is not 'teacher' or 'teacher's aide'
Clean Up	Select first name, last name, ~~and~~ title from ~~the~~ staff ~~table~~ where ~~the~~ title ~~is~~ not in ('teacher', ~~or~~ 'teacher''s aide')

SQL	`SELECT StfFirstName, StfLastName, Title`
	`FROM Staff`
	`WHERE NOT Title`
	`NOT IN ('Teacher', 'Teacher''s Aide')`

❖ **Note** We bet you're wondering about the two single quotes in the `'Teacher''s Aide'` character string literal. The SQL Standard dictates that you use a single quote to delimit a character string or datetime literal. When you need to embed a single quote within a character string literal, you must "clue in" your database system by entering the single quote twice. If you don't do that, the single quote acts as the end delimiter of the character string. The "`s Aide'`" that would occur after the second single quote would generate a syntax error!

We assume, of course, that one of the two NOT operators appears by mistake. You can still execute this SELECT statement, but it will send the wrong rows to the result set. In this case, the two NOT operators cancel each other—exactly like a double negative in arithmetic or in language—and the IN predicate now determines which rows are sent to the result set. So instead of seeing anyone *other than* a teacher or teacher's aide in the result set, you'll see *only* teachers and teacher's aides. Although you would not consciously define a search condition in this manner, you could very well do it accidentally. Remember that it's often the simple mistakes that cause the most problems.

Order of Precedence

The SQL Standard specifies how a database system should evaluate single conditions within a search condition and the order in which those evaluations take place. You've already learned in this chapter *how* a database evaluates each type of condition. Now we'll show you how the database determines *when* to evaluate each single condition.

By default, the database evaluates conditions from left to right. This is particularly true in the case of simple conditions. In the following example, the SELECT statement first searches for rows where the ship date is equal to the order date and then determines which of those rows contain customer number 1001. The rows that meet both conditions are then sent to the result set.

```
SQL         SELECT CustomerID, OrderDate, ShipDate
            FROM Orders
            WHERE ShipDate = OrderDate
               AND CustomerID = 1001
```

To have the SELECT statement search for a specific customer number before evaluating the ship date, just switch the position of the conditions. We'll discuss why you might want to do this later in this section.

When a search condition contains various types of single conditions, the database evaluates them in a specific order based on the operator used in each condition. The SQL Standard defines the following order of precedence for operator evaluation.

Evaluation Order	Type of Operator
1	Positive sign (+), negative sign (–)
2	Multiplication (*), division (/)
3	Addition (+), subtraction (–)
4	=, <>, <, >, <=, >=, BETWEEN, IN, LIKE, IS NULL
5	NOT
6	AND
7	OR

The following SELECT statement contains an example of the type of search condition that causes the database system to follow the order of precedence. In this case, the database performs the addition operation, executes the comparisons, and determines whether either condition has been met. Any row that meets either condition is then sent to the result set.

```
SQL         SELECT CustomerID, OrderDate,ShipDate
            FROM Orders
            WHERE CustomerID = 1001
             OR ShipDate = OrderDate + 4
```

Prioritizing Conditions

You can greatly increase the accuracy of your search conditions by understanding the order of precedence. This knowledge will help you formulate exactly the right condition for the request at hand. But you must be careful to avoid defining ambiguous conditions because they can produce unexpected results.

Let's use the following example to take a look at this potential problem.

```
SQL         SELECT CustFirstName, CustLastName, CustState,
               CustZipCode
            FROM Orders
            WHERE CustLastName = 'Patterson'
               AND CustState = 'CA'
               OR CustZipCode LIKE '%9'
```

In this instance, it's difficult to determine the true intent of the search condition because there are two ways you can interpret it.

1. You're looking for everyone named Patterson in the state of California *or* anyone with a ZIP Code that ends with a 9.
2. You're specifically looking for everyone named Patterson *and* anyone who lives in California or has a ZIP Code that ends with a 9.

If you have memorized the evaluation order table, you know that the first way is correct because your system should evaluate AND before OR. But are you always going to remember the evaluation sequence? You can avoid this ambiguity and make the search condition clearer by using parentheses to combine and prioritize certain conditions. For example, to follow the first interpretation of the search condition, you define the WHERE clause in this manner.

```
WHERE (CustLastName = 'Patterson' AND CustState = 'CA')
   OR CustZipCode LIKE '%9'
```

The parentheses ensure that the database analyzes and evaluates the two comparison conditions *before* it performs the same processes on the pattern match condition.

You could instead follow the second interpretation and define the WHERE clause in this manner.

```
WHERE CustLastName = 'Patterson' AND (CustState = 'CA'
   OR CustZipCode LIKE '%9')
```

In this case, the database analyzes and evaluates the first comparison condition *after* it performs those processes on the second comparison condition and the pattern match condition.

The idea of enclosing conditions in parentheses should be familiar to you by now. You learned how to do this when we discussed combining conditions earlier in this chapter. Now we're trying to emphasize that the placement of the parentheses can have a serious impact on the outcome of the search condition.

You can define any number of parenthetical conditions and even embed them as necessary. Similar to processing expressions, search conditions are processed left to right and then innermost to outermost *except* that when two or more conditions are at an equal level, the database system processes AND first and then OR. Here's how the database handles parenthetical search conditions.

- Parenthetical search conditions are processed before nonparenthetical search conditions.

- Two or more parenthetical search conditions are processed from left to right.
- Embedded parenthetical search conditions within a search condition are processed from innermost to outermost.

After the database begins to analyze a given parenthetical condition, it evaluates all expressions within the condition using the normal order of precedence. If you carefully translate your request and make effective use of parentheses within the search condition, you'll have better results.

Less Is Better Than More

We said at the beginning of this section that the database initially evaluates conditions from left to right and that it invokes the order of precedence when you define and use complex conditions. We also said that the manner in which you use parentheses in a search condition has a direct impact on its outcome. Now we'll pass along a simple, generic tip for speeding up the search condition process: Ask for less. That is, select only those columns you need to fulfill the request, and make the search condition as specific as you can so that your database processes the fewest rows possible. When you need to use multiple conditions, make certain that the condition that excludes the most rows from the result set is processed first so that your database can potentially find the answer faster. (Here's where your understanding of the order of precedence is really beneficial.)

We'll demonstrate this tip with an example we used earlier in this section.

```
SQL         SELECT CustomerID, OrderDate, ShipDate
            FROM Orders
            WHERE ShipDate = OrderDate
               AND CustomerID = 1001
```

In this instance, a row must fulfill both conditions in order for it to be included in the result set. Placing the predicates in this order tells your database to search for each ship date that is equal to its respective order date first. Depending on the number of rows in the table, it could take the database quite some time to evaluate this condition. Then the database will search the rows that met the first condition to identify which ones contain customer ID 1001.

Here's perhaps a better way to define the condition.

```
SQL         SELECT CustomerID, OrderDate, ShipDate
            FROM Orders
            WHERE CustomerID = 1001
               AND ShipDate = OrderDate
```

Now the database is more likely to search for the customer ID first. This condition is more likely to produce a small number of rows, which means that the database will need less time to search for the rows that match the ship date predicate.

You should make this technique a common practice and apply it when you define your search conditions. This will go a long way in helping to ensure that your SELECT statements execute quickly and efficiently. Be sure to study your database system's documentation to learn what other techniques you can apply to optimize the SELECT statement even further.

❖ **Note** Virtually all commercial database systems include a query optimizer that looks at your entire request and tries to figure out the fastest way to return the answer. The indexes that your database administrator has defined on columns in your tables have the biggest influence on what most optimizers choose to do. But it doesn't hurt to make it a practice to include the most exclusive search condition first to further influence your database system's optimizer.

Now that you understand combining search conditions, let's take a short side trip to something more complex. What do you do when you want to find rows that contain a range of values compared to another range of values? Read on!

Checking for Overlapping Ranges

BETWEEN works really well when you're looking for a value in a single column that is within a range of values. You also learned that you can test a single value to see whether it is within the range defined by a pair of start/end or low/high columns in your table. But what should you do if you want to find out whether one range overlaps with another? For example, you might want to know all the engagements (each has a start date and an end date) that occur any time during the week of November 12, 2007, through November 18, 2007. You might be tempted to solve the problem using BETWEEN like this:

"Show me the engagements that occur during the week of November 12, 2007, through November 18, 2007."

Translation	Select engagement number, start date, and end date from the engagements table where start date is between November 12, 2007, and November 18, 2007 and end date is between November 12, 2007, and November 18, 2007
Clean Up	Select engagement number, start date, ~~and~~ end date from ~~the~~ engagements ~~table~~ where start date ~~is~~ between ~~November 12, 2007~~ '2007-11-12' and ~~November 18, 2007~~ '2007-11-18' and end date ~~is~~ between ~~November 12, 2007~~ '2007-11-12' and ~~November 18, 2007~~ '2007-11-18'

SQL	`SELECT EngagementNumber, StartDate, EndDate`
	`FROM Engagements`
	`WHERE StartDate BETWEEN '2007-11-12' AND '2007-11-18'`
	`AND EndDate BETWEEN '2007-11-12' AND '2007-11-18'`

Close, but no cigar. You really want any engagement that has any date that falls between the two dates in November. To understand why a simple combination of BETWEEN clauses doesn't work, consider Figure 6–12.

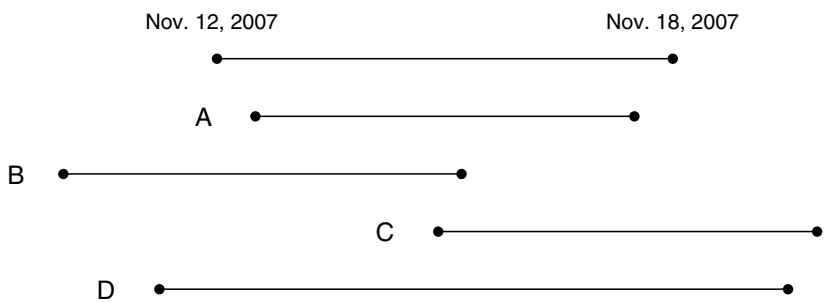


Figure 6–12 *Engagements that occur within the desired date span*

As you can see in the figure, there are four possible engagement date spans that can occur either entirely or partially within the week you want. Some engagements occur entirely within the date span, as represented by line A. Some start before the date span but end within the date span, as represented by line B. Others might start within the date span but end after the date span, as represented by line C. And finally, some engagements might start before the date span and not end until after the date span, as shown in line D.

If you think about the request as originally stated, the only engagements you'll find are those that are like line A. B gets excluded because the start date is not between November 12 and November 18 even though part of the engage-

ment occurs within the desired date span. C gets excluded because the end date is not between the two dates of interest. And D gets excluded because both the start and end dates are outside the range even though some dates of the engagement *do* occur entirely within the date span of interest.

So, how do you solve this problem? You explicitly create a search condition for each of the four possible scenarios, like this:

```
WHERE (StartDate BETWEEN '2007-11-12' AND '2007-11-18'
AND EndDate BETWEEN '2007-11-12' AND '2007-11-18')
OR (StartDate <= '2007-11-12')
AND EndDate BETWEEN '2007-11-12' AND '2007-11-18')
OR (StartDate BETWEEN '2007-11-12' AND '2007-11-18'
AND EndDate >= '2007-11-18')
OR (StartDate <= '2007-11-12'
AND EndDate >= '2007-11-18')
```

Not pretty, is it? But take a look at the figure again. What one thing do all the start dates have in common? They're all less than or equal to the end date of the span! Likewise, the end dates are all greater than or equal to the start date of the span. So the simple answer is as follows.

```
SQL         SELECT EngagementNumber, StartDate, EndDate
            FROM Engagements
            WHERE StartDate <= '2007-11-18'
            AND EndDate >= '2007-11-12'
```

Isn't that a lot simpler? Keep this solution in mind—you'll need it to solve one of the sample problems at the end of the chapter. Now back to our regular programming—let's revisit Nulls.

Nulls Revisited: A Cautionary Note

Now is as good a time as any to remind you about Nulls. You learned in Chapter 5 that a Null represents the absence of a value and that an expression processing a Null value will return a Null value. The same holds true for search conditions as well. A predicate that evaluates a Null value *can never be true*. This might seem confusing, but the predicate can never be false either! The SQL Standard defines the result of any predicate that evaluates a Null as *unknown*. Remember that a predicate must be true for a row to be selected, so a false or unknown result will reject the row.

To help clarify the matter, let's reexamine in Figures 6–13 and 6–14 (page 194) the truth tables we first showed you in Figures 6–9 and 6–10 (pages 180 and 181). But this time, let's include the unknown result you will get if a Null is involved.

First Expression \ Second Expression	True	False	Unknown
AND			
True	True (Rows are selected)	False (Rows are rejected)	Unknown (Rows are rejected)
False	False (Rows are rejected)	False (Rows are rejected)	False (Rows are rejected)
Unknown	Unknown (Rows are rejected)	False (Rows are rejected)	Unknown (Rows are rejected)

Figure 6–13 *The result of combining two predicate expressions with the AND operator when either expression is Null (unknown)*

First Expression \ Second Expression	True	False	Unknown
OR			
True	True (Rows are selected)	True (Rows are selected)	True (Rows are selected)
False	True (Rows are selected)	False (Rows are rejected)	Unknown (Rows are rejected)
Unknown	True (Rows are selected)	Unknown (Rows are rejected)	Unknown (Rows are rejected)

Figure 6–14 *The result of combining two predicate expressions with the OR operator when either expression is Null (unknown)*

You can see that an unknown result from evaluating a predicate on a Null column really throws a monkey wrench into the picture! For example, let's assume you have a simple comparison predicate: A = B. If either A or B for a given row is the Null value, then the result of the comparison is unknown. Because the result is not true, the row won't be selected. If A = B is not true, you might expect that NOT (A = B) would be true. No! This is unknown also. Figure 6–15 helps you understand how this is so.

(Expression)	NOT (Expression)
True	**False**
False	**True**
Unknown	**Unknown**

Figure 6–15 *The result of applying NOT to a true/false/unknown value*

Suppose you're making the following request to the database.

> *"Let me see the names and phone numbers of King County residents whose last names are Hernandez."*

Translation: Select first name, last name, and phone number from the customers table where the county name is 'King' and the last name is 'Hernandez'

Clean Up:
Select first name, last name, ~~and~~ phone number
from ~~the~~ customers ~~table~~
where ~~the~~ county ~~name is~~ = 'King'
and ~~the~~ last name ~~is~~ = 'Hernandez'

SQL:

```
SELECT CustFirstName, CustLastName, CustPhoneNumber
FROM Customers
WHERE CustCounty = 'King'
   AND CustLastName = 'Hernandez'
```

As you know, a row must meet *both* conditions to be included in the result set. If either the county name or the last name is Null, the database disregards the row completely.

Let's now consider this request.

"Show me the names of all staff members who are graduate counselors or were hired on September 1, 2007."

Translation: Select last name and first name from the staff table where the title is 'graduate counselor' or date hired is September 1, 2007

Clean Up:
Select last name ~~and~~ first name
from ~~the~~ staff ~~table~~
where ~~the~~ title ~~is~~ = 'graduate counselor'
or date hired ~~is~~ = ~~September 1,2007~~ '2007-09-01'

SQL:
```
SELECT StfLastName, StfFirstName
FROM Staff
WHERE Title = 'Graduate Counselor'
   OR DateHired = '2007-09-01'
```

Although you might expect Nulls to have the same effect on conditions combined with OR as they do on conditions combined with AND, that is not necessarily the case. A row still has a chance of being included in the result set as long as it meets *either* of these conditions. Take a look at Figure 6–14 (page 194) again. Based on the values of Title and DateHired, Table 6–2 shows how the database determines whether to send a row to the result set when you combine the predicates with OR.

Table 6–2 *Determining the Result Set with OR*

Value of Title	Value of DateHired	Result
Graduate Counselor	2007-09-01	The row is included in the result set because it meets both conditions.
Graduate Counselor	2007-11-15	The row is included in the result set because it meets the first condition.
Registrar	**2007-09-01**	The row is included in the result set because it meets the second condition.
Graduate Counselor	Null	The row is included in the result set because it meets the first condition.
Null	**2007-09-01**	The row is included in the result set because it meets the second condition.
Null	Null	The row is excluded from the result set because it does not meet either condition.

When you suspect that a result set is displaying incorrect information, test any columns you're using as criteria with the Null condition. This will give you the opportunity to deal with any Null values as appropriate, and you can then execute your original SELECT statement once again. For example, if you think there might be a few graduate counselors missing from the result set, you could execute the following SELECT statement to determine whether this is true.

```
SQL         SELECT StfLastName, StfFirstName, Title
            FROM Staff
            WHERE Title IS NULL
```

If there are Null values in the Title column, this SELECT statement will produce a result set that contains the names of all staff members who do not have a title specified in the database. Now you can deal with this data as appropriate and then return to your original SELECT statement.

We're not done dealing with Nulls just yet. We'll revisit Nulls once more in Chapter 12, Simple Totals, when we discuss SELECT statements that summarize data.

Expressing Conditions in Different Ways

One side benefit to everything you've learned in this chapter is that you now have the ability to express a given condition in various ways. Let's take a look at this by considering the following request.

"Give me the name of every employee who was hired in October 2007."

You need to search for hire dates that fall between October 1, 2007, and October 31, 2007, in order to answer this request. Based on what you've already learned, you can define the condition in two ways.

```
DateHired BETWEEN '2007-10-01' AND '2007-10-31'
DateHired >= '2007-10-01' AND DateHired <= '2007-10-31'
```

Both of these conditions will send the same rows to the result set—the condition you choose to use is only a matter of preference. Some people find the first expression easier to understand, although others prefer the second expression.

Here are some other examples of equivalent conditions.

"Show me the vendors who are based in California, Oregon, or Washington."

```
VendState IN ('CA', 'OR', 'WA')
VendState = 'CA' OR VendState = 'OR' OR VendState = 'WA'
```

"Give me a list of customers whose last name begins with 'H'."

```
CustLastName >= 'H' AND CustLastName <= 'HZ'
CustLastName BETWEEN 'H' AND 'HZ'
CustLastName LIKE 'H%'
```

"Show me all the students who do not live in Seattle or Redmond."

```
StudCity <> 'Seattle' AND StudCity <> 'Redmond'
StudCity NOT IN ('Seattle', 'Redmond')
NOT (StudCity = 'Seattle' OR StudCity = 'Redmond')
```

There's no wrong way for you to define a condition, but you can define a condition incorrectly by blatantly disregarding its syntax. (As you know, this will cause the condition to fail.) However, some database systems optimize certain types of conditions for speedy processing, making them preferable to other equivalent conditions. Check your database system's documentation to determine whether your system has any preferred methods for defining conditions.

Sample Statements

You've now learned all the techniques you need to build solid search conditions. Let's take a look at some examples of various types of search conditions using the tables from each of the sample databases. These examples illustrate the use of search conditions to filter your data.

We've also included sample result sets that would be returned by these operations and placed them immediately after the SQL syntax line. The name that appears immediately above a result set is the name we gave each query in the sample data on the companion CD you'll find bound into the back of the book. We stored each query in the appropriate sample database (as indicated within the example) and prefixed the names of the queries relevant to this chapter with "CH06." You can follow the instructions in the Introduction of this book to load the samples onto your computer and try them.

❖ **Note** We've combined the Translation and Clean Up steps for all the examples once again so that you can continue to learn how to consolidate the process.

Sales Orders Database

"Show me all the orders for customer number 1001."

Translation/ Clean Up: Select ~~the~~ order number ~~and~~ customer ID from ~~the~~ orders ~~table~~ where ~~the~~ customer ID ~~is equal to~~ = 1001

SQL:

```
SELECT OrderNumber, CustomerID
FROM Orders
WHERE CustomerID = 1001
```

CH06_Orders_for_Customer_1001 (44 Rows)

OrderNumber	CustomerID
2	1001
7	1001
16	1001
52	1001
55	1001
107	1001
137	1001
138	1001
151	1001
154	1001
<< more rows here >>	

"Show me an alphabetized list of products with names that begin with 'Dog'."

Translation/ Clean Up
Select ~~the~~ product name from ~~the~~ products ~~table~~ where ~~the~~ product name like 'Dog%'
~~and~~ order by product name

SQL

```
SELECT ProductName
FROM Products
WHERE ProductName LIKE 'Dog%'
ORDER BY ProductName
```

CH06_Products_That_Begin_With_DOG (4 Rows)

ProductName
Dog Ear Aero-Flow Floor Pump
Dog Ear Cyclecomputer
Dog Ear Helmet Mount Mirrors
Dog Ear Monster Grip Gloves

❖ **Note** We just wanted to remind you that you place the ORDER BY clause at *the end* of a SELECT statement. If necessary, review the Sorting Information section in Chapter 4.

Entertainment Agency Database

"Show me an alphabetical list of entertainers based in Bellevue, Redmond, or Woodinville."

Translation/ Clean Up
Select stage name, phone number, ~~and~~ city
from ~~the~~ entertainers ~~table~~
where ~~the~~ city ~~is~~ in ('Bellevue', 'Redmond', ~~or~~ 'Woodinville')
~~and~~ order by stage name

SQL

```
SELECT EntStageName, EntPhoneNumber, EntCity
FROM Entertainers
WHERE EntCity
IN ('Bellevue', 'Redmond', 'Woodinville')
ORDER BY EntStageName
```

CH06_Eastside_Entertainers (7 Rows)

EntStageName	EntPhoneNumber	EntCity
Carol Peacock Trio	555-2691	Redmond
Jazz Persuasion	555-2541	Bellevue
Jim Glynn	555-2531	Bellevue
JV & the Deep Six	555-2511	Redmond
Katherine Ehrlich	555-0399	Woodinville
Modern Dance	555-2631	Woodinville
Susan McLain	555-2301	Bellevue

"Show me all the engagements that run for four days."

Translation/ Clean Up: Select engagement number, start date, ~~and~~ end date
from ~~the~~ engagements ~~table~~
where ~~the~~ CAST(end date ~~minus~~ – start date AS INTEGER) ~~is equal to~~ = 3

SQL:

```
SELECT EngagementNumber, StartDate, EndDate
FROM Engagements
WHERE CAST(EndDate – StartDate AS INTEGER) = 3
```

CH06_Four-Day Engagements (15 Rows)

EngagementNumber	StartDate	EndDate
5	2007-09-11	2007-09-14
13	2007-09-17	2007-09-20
17	2007-09-29	2007-10-02
21	2007-09-30	2007-10-03
56	2007-11-25	2007-11-28
58	2007-12-01	2007-12-04
59	2007-12-01	2007-12-04
63	2007-12-18	2007-12-21
70	2007-12-23	2007-12-26
95	2008-01-15	2008-01-18
<< more rows here >>		

❖ **Note** An engagement runs from the start date *through* the end date. When subtracting StartDate from EndDate, we get one less day than the total number of days for the engagement. For this reason, we compared the result of the calculation to 3, not 4.

School Scheduling Database

"Show me an alphabetical list of all the staff members and their salaries if they make between $40,000 and $50,000 a year."

Translation / Select first name, last name, ~~and~~ salary

Clean Up from ~~the~~ staff ~~table~~
where ~~the~~ salary ~~is~~ between 40000 and 50000~~, then~~
order by last name, ~~and~~ first name

SQL
```
SELECT StfFirstName, StfLastName, Salary
FROM Staff
WHERE Salary BETWEEN 40000 AND 50000
ORDER BY StfLastname, StfFirstName
```

CH06_Staff_Salaries_40K_TO_50K (14 Rows)

StfFirstName	StfLastName	Salary
Robert	Brown	$49,000.00
Kirk	DeGrasse	$45,000.00
Katherine	Ehrlich	$45,000.00
Jim	Glynn	$45,000.00
Liz	Keyser	$48,000.00
Ann	Patterson	$45,000.00
Maria	Patterson	$48,000.00
Mariya	Sergienko	$45,000.00
Tim	Smith	$40,000.00
Caleb	Viescas	$45,000.00
<< more rows here >>		

"Show me a list of students whose last name is 'Kennedy' or who live in Seattle."

Translation/ Clean Up

Select first name, last name, ~~and~~ city
from ~~the~~ students ~~table~~
where ~~the~~ last name ~~is~~ = 'Kennedy'
or ~~the~~ city ~~is~~ = 'Seattle'

SQL

```
SELECT StudFirstName, StudLastName, StudCity
FROM Students
WHERE StudLastName = 'Kennedy'
OR StudCity = 'Seattle'
```

Seattle_Students_And_Students_Named_Kennedy (4 Rows)

StudFirstName	StudLastName	StudCity
Doris	Hartwig	Seattle
John	Kennedy	Portland
Kendra	Bonnicksen	Seattle
Richard	Lum	Seattle

Bowling League Database

"List the ID numbers of the teams that won one or more of the first ten matches in Game 3."

Translation / Clean Up: Select ~~the~~ team ID, match ID, ~~and~~ game number from ~~the~~ match_games ~~table~~ where ~~the~~ game number ~~is~~ = 3 and ~~the~~ match ID ~~is~~ between 1 and 10

SQL:

```
SELECT WinningTeamID, MatchID, GameNumber
FROM Match_Games
WHERE GameNumber = 3 AND MatchID BETWEEN 1 AND 10
```

Game3_Top_Ten_Matches (10 Rows)

WinningTeamID	MatchID	GameNumber
1	1	3
3	2	3
5	3	3
7	4	3
3	5	3
4	6	3
5	7	3
8	8	3
2	9	3
1	10	3

"List the bowlers in teams 3, 4, and 5 whose last names begin with the letter 'H'."

Translation / Clean Up: Select first name, last name, ~~and~~ team ID from ~~the~~ bowlers ~~table~~ where ~~the~~ team ID ~~is either~~ in (3, 4, ~~or~~ 5) and ~~the~~ last name ~~begins with the letter~~ like 'H%'

SQL:

```
SELECT BowlerFirstName, BowlerLastName, TeamID
   FROM Bowlers
   WHERE (TeamID IN (3,4,5))
   AND (BowlerLastName LIKE 'H%')
```

H_Bowlers_Teams_3_Through_5 (4 Rows)

BowlerFirstName	BowlerLastName	TeamID
Elizabeth	Hallmark	4
Gary	Hallmark	4
Kendra	Hernandez	5
Michael	Hernandez	5

Recipes Database

"List the recipes that have no notes."

Translation/ Clean Up: Select ~~the~~ recipe title from ~~the~~ recipes ~~table~~ where notes is ~~empty~~ Null

SQL:

```
SELECT RecipeTitle FROM Recipes
WHERE Notes IS NULL
```

CH06_Recipes_With_No_Notes (6 rows)

RecipeTitle
Irish Stew
Salsa Buena
Fettuccini Alfredo
Mike's Summer Salad
Roast Beef
Yorkshire Pudding

"Show the ingredients that are meats (ingredient class is 2) but that aren't chicken."

Translation/ Clean Up: Select ingredient name from ~~the~~ ingredients ~~table~~ where ingredient class ID ~~is equal to~~ = 2 and ingredient name ~~does~~ not ~~contain~~ like '%chicken%'

SQL:

```
SELECT IngredientName FROM Ingredients
   WHERE (IngredientClassID = 2)
   AND (IngredientName NOT LIKE '%chicken%')
```

CH06_Meats_That_Are_Not_Chicken (5 rows)

IngredientName
Beef
Bacon
T-bone Steak
New York Steak
Ground Pork

SUMMARY

In this chapter, we introduced you to the idea of filtering the information you see in a result set by using a search condition in a WHERE clause. You learned that a search condition uses combinations of predicates to filter the data sent to the result set and that predicates are specific tests you can apply to a value expression. We then introduced you to the five basic types of predicates.

Our discussion continued with an in-depth look at each of the five basic types of predicates you can define within a search condition of a WHERE clause. You learned how to compare values and how to test whether a value falls within a specified range of values. You also learned how to test whether a value matches one of a defined list of values or is part of a specific pattern string. Additionally, you learned that you could use the NOT operator to exclude rows from a result set.

We then discussed how to use multiple conditions by combining them with AND and OR operators. You learned that a row must meet all conditions

combined with AND before it can be included in the result set, whereas it must meet *only one* of those conditions if the conditions are combined with OR. You also learned how to use AND and OR together to answer complex requests. We then took a second look at using NOT to exclude rows from a result set, and you learned that NOT can be used at two different levels in a search condition.

The order of precedence was the next topic of discussion, and you learned how the database analyzes and evaluates conditions. You now know that the database evaluates conditions in a specific order based on the operator used in each condition. You also learned how to use parentheses to alter the order in which the database evaluates certain conditions and to ensure that you avoid defining ambiguous conditions.

We took a brief detour to show you how to search for a range across another range. The answer is surprisingly simple, and it doesn't involve using BETWEEN.

We next took another look at Nulls. Here you learned that Nulls affect conditions in much the way that they affect expressions. You also know that you should test for Null values if you suspect that a result set is displaying incorrect information.

Finally, we discussed the fact that the same condition can be expressed in various ways. You now know, for example, that you can use three different types of conditions to search for people whose last names begin with the letter "H."

In the next part of the book, we'll introduce you to the idea of *sets* and the types of operations you can perform on them. After you learn about sets, you'll be well on your way to learning how to define SELECT statements using multiple tables.

The following section presents a number of requests that you can work out on your own.

Problems for You to Solve

Below, we show you the request statement and the name of the solution query in the sample databases. If you want some practice, you can work out the SQL you need for each request and then check your answer with the query we saved in the samples. Don't worry if your syntax doesn't exactly match the syntax of the queries we saved—as long as your result set is the same.

Sales Orders Database

1. *"Give me the names of all vendors based in Ballard, Bellevue, and Redmond."*
   You can find the solution in CH06_Ballard_Bellevue_Redmond_Vendors (3 rows).
2. *"Show me an alphabetized list of products with a retail price of $125.00 or more."*
   (Hint: You'll alphabetize the list using a clause we discussed in a previous chapter.)
   You can find the solution in CH06_Products_Priced_Over_125 (13 rows).
3. *"Which vendors do we work with that don't have a Web site?"*
   You can find the solution in CH06_Vendors_With_No_Website (4 rows).

Entertainment Agency Database

1. *"Let me see a list of all engagements that occurred during October 2007."*
   (Hint: You need to solve this problem by testing for values in a range in the table that contain any values in another range—the first and last dates in October.)
   You can find the solution in CH06_October_2007_Engagements (23 rows).
2. *"Show me any engagements in October 2007 that start between noon and 5 P.M."*
   You can find the solution in CH06_October_Dates_Between_Noon_and_Five (17 rows).
3. *"List all the engagements that start and end on the same day."*
   You can find the solution in CH06_Single_Day_Engagements (5 rows).

School Scheduling Database

1. *"Show me which staff members use a post office box as their address."*
   You can find the solution in CH06_Staff_Using_POBoxes (3 rows).
2. *"Can you show me which students live outside of the Pacific Northwest?"*
   You can find the solution in CH06_Students_Residing_Outside_PNW (5 rows).
3. *"List all the subjects that have a subject code starting 'MUS'."*
   You can find the solution in CH06_Subjects_With_MUS_In_SubjectCode (4 rows).

Bowling League Database

1. *"Give me a list of the tournaments held during September 2007."*
   You can find the solution in CH06_September_2007_Tournament_Schedule (4 rows).
2. *"What are the tournament schedules for Bolero, Red Rooster, and Thunderbird Lanes?"*
   You can find the solution in CH06_Eastside_Tournaments (6 rows).
3. *"List the bowlers who live on the Eastside (you know—Bellevue, Bothell, Duvall, Redmond, and Woodinville) and who are on teams 5, 6, 7, or 8."*
   (Hint: Use IN for the city list and BETWEEN for the team numbers.)
   You can find the solution in CH06_Eastside_Bowlers_On_Teams_5_Through_8 (9 rows).

Recipes Database

1. *"List all recipes that are main courses (recipe class is 1) and that have notes."*
   You can find the solution in CH06_Main_Courses_With_Notes (4 rows).
2. *"Display the first five recipes."*
   (Hint: Use BETWEEN on the primary key of the table.)
   You can find the solution in CH06_First_5_Recipes (5 rows).

This page intentionally left blank

Part III

Working with Multiple Tables

This page intentionally left blank

7

Thinking in Sets

"Small cheer and a great welcome makes a merry feast."
—William Shakespeare
Comedy of Errors, Act 3, scene 1

Topics Covered in This Chapter

What Is a Set, Anyway?

Operations on Sets

Intersection

Difference

Union

SQL Set Operations

Summary

By now, you know how to create a set of information by asking for specific columns or expressions on columns (SELECT), how to sort the rows (ORDER BY), and how to restrict the rows returned (WHERE). Up to this point, we've been focusing on basic exercises involving a single table. But what if you want to know something about information contained in multiple tables? What if you want to compare or contrast sets of information from the same or different tables?

Creating a meal by peeling, slicing, and dicing a single pile of potatoes or a single bunch of carrots is easy. From here on out, most of the problems we're going to show you how to solve will involve getting data from *multiple* tables. We're not only going to show you how to put together a good stew—we're going to teach you how to be a chef!

Before digging into this chapter, you need to know that it's all about the *concepts* you must understand in order to successfully link two or more sets of

information. We're also going to give you a brief overview of some specific syntax defined in the SQL Standard that directly supports the pure definition of these concepts. Be forewarned, however, that many current commercial implementations of SQL do not yet support this "pure" syntax. In later chapters, we'll show you how to implement the concepts you'll learn here using SQL syntax that is commonly supported by most major database systems. What we're after here is not the letter of the law but rather the spirit of the law.

What Is a Set, Anyway?

If you were a teenager any time from the mid-1960s onward, you might have studied set theory in a mathematics course. (Remember New Math?) If you were introduced to set algebra, you probably wondered why any of it would ever be useful.

Now you're trying to learn about relational databases and this quirky language called SQL to build applications, solve problems, or just get answers to your questions. Were you paying attention in algebra class? If so, solving problems—particularly complex ones—in SQL will be much easier.

Actually, you've been working with sets from the beginning of this book. In Chapter 1, What Is Relational?, you learned about the basic structure of a relational database—tables containing records that are made up of one or more fields. (Remember that in SQL, records are known as rows, and fields are known as columns.) Each table in your database is a *set* of information about one subject. In Chapter 2, Ensuring Your Database Structure Is Sound, you learned how to verify that the structure of your database is sound. Each table should contain the *set* of information related to one and only one subject or action.

In Chapter 4, Creating a Simple Query, you learned how to build a basic SELECT statement in SQL to retrieve a result *set* of information that contains specific columns from a single table and how to sort those result sets. In Chapter 5, Getting More Than Simple Columns, you learned how to glean a new *set* of information from a table by writing expressions that operate on one or more columns. In Chapter 6, Filtering Your Data, you learned how to restrict further the *set* of information you retrieve from your tables by adding a filter (WHERE clause) to your query.

As you can see, a set can be as little as the data from one column from one row in one table. Actually, you can construct a request in SQL that returns no rows—an empty set. Sometimes it's useful to discover that something does

not exist. A set can also be multiple columns (including columns you create with expressions) from multiple rows fetched from multiple tables. Each row in a result set is a *member* of the set. The values in the columns are specific *attributes* of each member—data items that describe the member of the set. In the next several chapters, we'll show how to ask for information from multiple sets of data and link these sets together to get answers to more complex questions. First, however, you need to understand more about sets and the logical ways to combine them.

Operations on Sets

In Chapter 1, we discussed how Dr. E. F. Codd invented the relational model on which most modern databases and SQL are based. Two branches of mathematics—set theory and first-order predicate logic—formed the foundation of his new model.

After you graduate beyond getting answers from only a single table, you need to learn how to use result sets of information to solve more complex problems. These complex problems usually require using one of the common set operations to link data from two or more tables. Sometimes, you'll need to get two different result sets from the same table and then combine them to get your answer.

The three most common set operations are as follows.

- **Intersection**—You use this to find the common elements in two or more different sets: "Show me the recipes that contain *both* lamb *and* rice." "Show me the customers who ordered *both* bicycles *and* helmets."
- **Difference**—You use this to find items that are in one set but not another: "Show me the recipes that contain lamb but *do not* contain rice." "Show me the customers who ordered a bicycle but *not* a helmet."
- **Union**—You use this to combine two or more similar sets: "Show me all the recipes that contain *either* lamb *or* rice." "Show me the customers who ordered *either* a bicycle *or* a helmet."

In the following three sections, we'll explain these basic set operations—the ones you should have learned in high school algebra. The SQL Set Operations section later in this chapter gives an overview of how these operations are implemented in "pure" SQL.

Intersection

No, it's not your local street corner. An *intersection* of two sets contains the common elements of two sets. Let's first take a look at an intersection from the pure perspective of set theory and then see how you can use an intersection to solve business problems.

Intersection in Set Theory

An intersection is a very powerful mathematical tool often used by scientists and engineers. As a scientist, you might be interested in finding common points between two sets of chemical or physical sample data. For example, a pharmaceutical research chemist might have two compounds that seem to provide a certain beneficial effect. Finding the commonality (the intersection) between the two compounds might help discover what it is that makes the two compounds effective. Or, an engineer might be interested in finding the intersection between one alloy that is hard but brittle and another alloy that is soft but resilient.

Let's take a look at intersection in action by examining two sets of numbers. In this example, each single number is a member of the set. The first set of numbers is as follows.

1, 5, 8, 9, 32, 55, 78

The second set of numbers is as follows.

3, 7, 8, 22, 55, 71, 99

The intersection of these two sets of numbers is the numbers common to both sets.

8, 55

The individual entries—the members—of each set don't have to be just single values. In fact, when solving problems with SQL, you'll probably deal with sets of rows.

According to set theory, when a member of a set is something more than a single number or value, each member (or object) of the set has multiple attributes or bits of data that describe the properties of each member. For

example, your favorite stew recipe is a complex member of the set of all recipes that contains many different ingredients. Each ingredient is an attribute of your complex stew member.

To find the intersection between two sets of complex set members, you have to find the members that match on all the attributes. Also, all the members in each set you're trying to compare must have the same number and type of attributes. For example, suppose you have a complex set like the one below, in which each row represents a member of the set (a stew recipe), and each column denotes a particular attribute (an ingredient).

Potatoes	Water	Lamb	Peas
Rice	Chicken Stock	Chicken	Carrots
Pasta	Water	Tofu	Snap Peas
Potatoes	Beef Stock	Beef	Cabbage
Pasta	Water	Pork	Onions

A second set might look like the following.

Potatoes	Water	Lamb	Onions
Rice	Chicken Stock	Turkey	Carrots
Pasta	Vegetable Stock	Tofu	Snap Peas
Potatoes	Beef Stock	Beef	Cabbage
Beans	Water	Pork	Onions

The intersection of these two sets is the one member whose attributes all match in both sets.

Potatoes	Beef Stock	Beef	Cabbage

Intersection between Result Sets

If the previous examples look like rows in a table or a result set to you, you're on the right track! When you're dealing with rows in a set of data that you

fetch with SQL, the attributes are the individual columns. For example, suppose you have a set of rows returned by a query like the following one. (These are recipes from John's cookbook.)

Recipe	Starch	Stock	Meat	Vegetable
Lamb Stew	Potatoes	Water	Lamb	Peas
Chicken Stew	Rice	Chicken Stock	Chicken	Carrots
Veggie Stew	Pasta	Water	Tofu	Snap Peas
Irish Stew	Potatoes	Beef Stock	Beef	Cabbage
Pork Stew	Pasta	Water	Pork	Onions

A second query result set might look like the following. (These are recipes from Mike's cookbook.)

Recipe	Starch	Stock	Meat	Vegetable
Lamb Stew	Potatoes	Water	Lamb	Peas
Turkey Stew	Rice	Chicken Stock	Turkey	Carrots
Veggie Stew	Pasta	Vegetable Stock	Tofu	Snap Peas
Irish Stew	Potatoes	Beef Stock	Beef	Cabbage
Pork Stew	Beans	Water	Pork	Onions

The intersection of these two sets is the two members whose attributes all match in both sets—that is, the two recipes that Mike and John have in common.

Recipe	Starch	Stock	Meat	Vegetable
Lamb Stew	Potatoes	Water	Lamb	Peas
Irish Stew	Potatoes	Beef Stock	Beef	Cabbage

Sometimes it's easier to see how intersection works using a set diagram. A *set diagram* is an elegant yet simple way to diagram sets of information and

graphically represent how the sets intersect or overlap. You might also have heard this sort of diagram called a Euler or Venn diagram. (By the way, Leonard Euler was an eighteenth-century Swiss mathematician, and John Venn used this particular type of logic diagram in 1880 in a paper he wrote while a Fellow at Cambridge University. So you can see that "thinking in sets" is not a particularly modern concept!)

Let's assume you have a nice database containing all your favorite recipes. You really like the way onions enhance the flavor of beef, so you're interested in finding all recipes that contain both beef and onions. Figure 7–1 shows the set diagram that helps you visualize how to solve this problem.

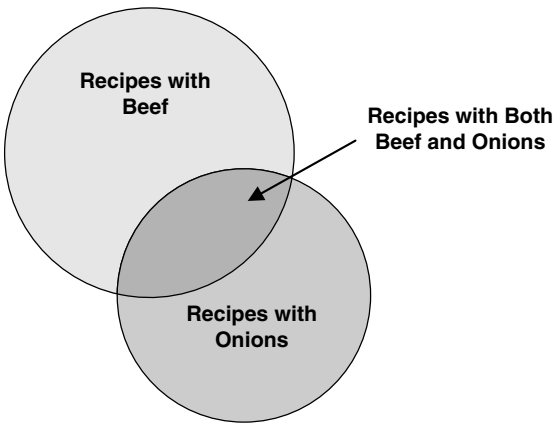


Figure 7–1 *Finding out which recipes have both beef and onions*

The upper circle represents the set of recipes that contain beef. The lower circle represents the set of recipes that contain onions. Where the two circles overlap is where you'll find the recipes that contain both—the intersection of the two sets. As you can imagine, you first ask SQL to fetch all the recipes that have beef. In the second query, you ask SQL to fetch all the recipes that have onions. As you'll see later, you can use a special SQL keyword—INTERSECT—to link the two queries to get the final answer.

Yes, we know what you're thinking. If your recipe table looks like the samples above, you could simply say the following.

"Show me the recipes that have beef as the meat ingredient and onions as the vegetable ingredient."

Translation	Select the recipe name from the recipes table where meat ingredient is beef and vegetable ingredient is onions
Clean Up	Select ~~the~~ recipe name from ~~the~~ recipes ~~table~~ where meat ingredient ~~is~~ = beef and vegetable ingredient ~~is~~ = onions

SQL

```
SELECT RecipeName
FROM Recipes
WHERE MeatIngredient = 'Beef'
   AND VegetableIngredient = 'Onions'
```

Hold on now! If you remember the lessons you learned in Chapter 2, you know that a single Recipes table probably won't cut it. (Pun intended!) What about recipes that have ingredients other than meat and vegetables? What about the fact that some recipes have many ingredients and others have only a few? A correctly designed recipes database will have a separate Recipe_Ingredients table with one row per recipe per ingredient. Each ingredient row will have only one ingredient, so no single row can be both beef and onions at the same time. You'll need to first find all the beef rows, then find all the onions rows, and then intersect them on RecipeID. (If you're confused about why we're criticizing the previous table design, be sure to go back and read Chapter 2!)

How about a more complex problem? Let's say you want to add carrots to the mix. A set diagram to visualize the solution might look like Figure 7–2.

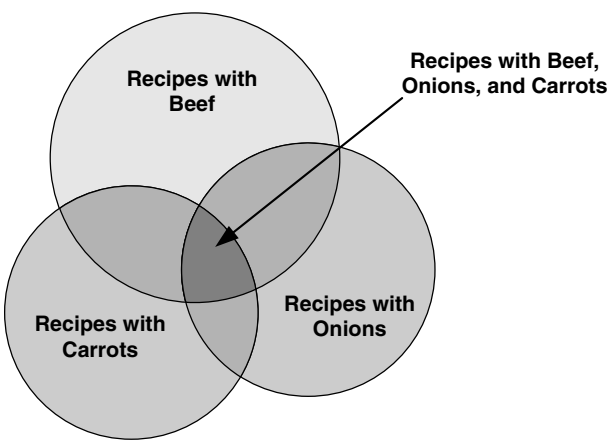


Figure 7–2 *Determining which recipes have beef, onions, and carrots*

Got the hang of it? The bottom line is that when you're faced with solving a problem involving complex criteria, a set diagram can be an invaluable way to see the solution expressed as the intersection of SQL result sets.

Problems You Can Solve with an Intersection

As you might guess, you can use an intersection to find the matches between two or more sets of information. Here's just a small sample of the problems you can solve using an intersection technique with data from the sample databases.

"Show me customers and employees who have the same name."

"Find all the customers who ordered a bicycle and also ordered a helmet."

"List the entertainers who played engagements for customers Bonnicksen and Rosales."

"Show me the students who have an average score of 85 or better in Art and who also have an average score of 85 or better in Computer Science."

"Find the bowlers who had a raw score of 155 or better at both Thunderbird Lanes and Bolero Lanes."

"Show me the recipes that have beef and garlic."

One of the limitations of using a pure intersection is that the values must match in all the columns in each result set. This works well if you're intersecting two or more sets from the same table—for example, customers who ordered bicycles and customers who ordered helmets. It also works well when you're intersecting sets from tables that have similar columns—for example, customer names and employee names. In many cases, however, you'll want to find solutions that require a match on only a few column values from each set. For this type of problem, SQL provides an operation called a JOIN—an intersection on key values. Here's a sample of problems you can solve with a JOIN.

"Show me customers and employees who live in the same city." (JOIN on the city name.)

"List customers and the entertainers they booked." (JOIN on the engagement number.)

"Find the agents and entertainers who live in the same ZIP Code." (JOIN on the ZIP Code.)

"Show me the students and their teachers who have the same first name." (JOIN on the first name.)

"Find the bowlers who are on the same team." (JOIN on the team ID.)

"Display all the ingredients for recipes that contain carrots." (JOIN on the ingredient ID.)

Never fear. In the next chapter we'll show you all about solving these problems (and more) by using JOINs. And because so few commercial implementations of SQL support INTERSECT, we'll show how to use a JOIN to solve many problems that might otherwise require an INTERSECT.

Difference

What's the difference between 21 and 10? If you answered 11, you're on the right track! A *difference* operation (sometimes also called subtract, minus, or except) takes one set of values and removes the set of values from a second set. What remains is the set of values in the first set that are *not* in the second set. (As you'll see later, EXCEPT is the keyword used in the SQL Standard.)

Difference in Set Theory

Difference is another very powerful mathematical tool. As a scientist, you might be interested in finding what's different about two sets of chemical or physical sample data. For example, a pharmaceutical research chemist might have two compounds that seem to be very similar, but one provides a certain beneficial effect and the other does not. Finding what's different about the two compounds might help uncover why one works and the other does not. As an engineer, you might have two similar designs, but one works better than the other. Finding the difference between the two designs could be crucial to eliminating structural flaws in future buildings.

Let's take a look at difference in action by examining two sets of numbers. The first set of numbers is as follows.

1, 5, 8, 9, 32, 55, 78

The second set of numbers is as follows.

3, 7, 8, 22, 55, 71, 99

The difference of the first set of numbers minus the second set of numbers is the numbers that exist in the first set but not the second.

1, 5, 9, 32, 78

Note that you can turn the previous difference operation around. Thus, the difference of the second set minus the first set is

3, 7, 22, 71, 99

The members of each set don't have to be single values. In fact, you'll most likely be dealing with sets of rows when trying to solve problems with SQL.

Earlier in this chapter we said that when a member of a set is something more than a single number or value, each member of the set has multiple attributes (bits of information that describe the properties of each member). For example, your favorite stew recipe is a complex member of the set of all recipes that contains many different ingredients. You can think of each ingredient as an attribute of your complex stew member.

To find the difference between two sets of complex set members, you have to find the members that match on all the attributes in the second set with members in the first set. Don't forget that all of the members in each set you're trying to compare must have the same number and type of attributes. Remove from the first set all the matching members you find in the second set, and the result is the difference. For example, suppose you have a complex set like the one below. Each row represents a member of the set (a stew recipe), and each column denotes a particular attribute (an ingredient).

Potatoes	Water	Lamb	Peas
Rice	Chicken Stock	Chicken	Carrots
Pasta	Water	Tofu	Snap Peas
Potatoes	Beef Stock	Beef	Cabbage
Pasta	Water	Pork	Onions

A second set might look like this.

Potatoes	Water	Lamb	Onions
Rice	Chicken Stock	Turkey	Carrots
Pasta	Vegetable Stock	Tofu	Snap Peas
Potatoes	Beef Stock	Beef	Cabbage
Beans	Water	Pork	Onions

The difference of the first set minus the second set is the objects in the first set that don't exist in the second set.

Potatoes	Water	Lamb	Peas
Rice	Chicken Stock	Chicken	Carrots
Pasta	Water	Tofu	Snap Peas
Pasta	Water	Pork	Onions

Difference between Result Sets

When you're dealing with rows in a set of data fetched with SQL, the attributes are the individual columns. For example, suppose you have a set of rows returned by a query like the following one. (These are recipes from John's cookbook.)

Recipe	Starch	Stock	Meat	Vegetable
Lamb Stew	Potatoes	Water	Lamb	Peas
Chicken Stew	Rice	Chicken Stock	Chicken	Carrots
Veggie Stew	Pasta	Water	Tofu	Snap Peas
Irish Stew	Potatoes	Beef Stock	Beef	Cabbage
Pork Stew	Pasta	Water	Pork	Onions

A second query result set might look like the following. (These are recipes from Mike's cookbook.)

Recipe	Starch	Stock	Meat	Vegetable
Lamb Stew	Potatoes	Water	Lamb	Peas
Turkey Stew	Rice	Chicken Stock	Turkey	Carrots
Veggie Stew	Pasta	Vegetable Stock	Tofu	Snap Peas
Irish Stew	Potatoes	Beef Stock	Beef	Cabbage
Pork Stew	Beans	Water	Pork	Onions

The difference between John's recipes and Mike's recipes (John's minus Mike's) is all the recipes in John's cookbook that *do not* appear in Mike's cookbook.

Recipe	Starch	Stock	Meat	Vegetable
Chicken Stew	Rice	Chicken Stock	Chicken	Carrots
Veggie Stew	Pasta	Water	Tofu	Snap Peas
Pork Stew	Pasta	Water	Pork	Onions

You can also turn this problem around. Suppose you want to find the recipes in Mike's cookbook that *are not* in John's cookbook. Here's the answer.

Recipe	Starch	Stock	Meat	Vegetable
Turkey Stew	Rice	Chicken Stock	Turkey	Carrots
Veggie Stew	Pasta	Vegetable Stock	Tofu	Snap Peas
Pork Stew	Beans	Water	Pork	Onions

Again, we can use a set diagram to help visualize how a difference operation works. Let's assume you have a nice database containing all your favorite recipes. You really do not like the way onions taste with beef, so you're interested in finding all recipes that contain beef but not onions. Figure 7–3 shows you the set diagram that helps you visualize how to solve this problem.

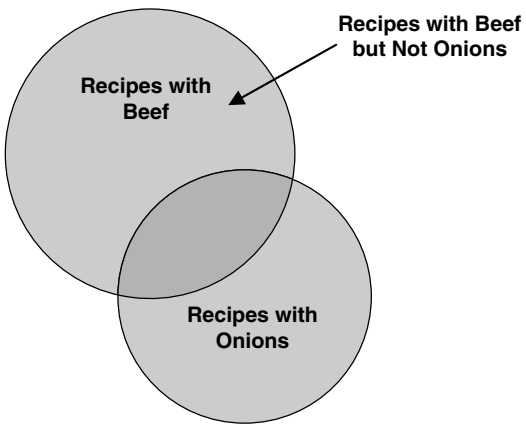


Figure 7–3 *Finding out which recipes have beef but not onions*

The upper full circle represents the set of recipes that contain beef. The lower full circle represents the set of recipes that contain onions. As you remember from the discussion about INTERSECT, where the two circles overlap is where you'll find the recipes that contain both. The dark-shaded part of the upper circle that's not part of the overlapping area represents the set of recipes that contain beef but do not contain onions. Likewise, the part of the lower circle that's not part of the overlapping area represents the set of recipes that contain onions but do not contain beef.

You probably know that you first ask SQL to fetch all the recipes that have beef. Next, you ask SQL to fetch all the recipes that have onions. (As you'll see later in this chapter, the special SQL keyword EXCEPT links the two queries to get the final answer.)

Are you falling into the trap again? (You *did* read Chapter 2, didn't you?) If your recipe table looks like the samples earlier, you might think that you could simply say the following.

"Show me the recipes that have beef as the meat ingredient and that do not have onions as the vegetable ingredient."

Translation: Select the recipe name from the recipes table where meat ingredient is beef and vegetable ingredient is not onions

Clean Up: Select ~~the~~ recipe name from ~~the~~ recipes ~~table~~ where meat ingredient ~~is~~ = beef and vegetable ingredient ~~is not~~ <> onions

SQL:

```
SELECT RecipeName
FROM Recipes
WHERE MeatIngredient = 'Beef'
   AND VegetableIngredient <> 'Onions'
```

Again, as you learned in Chapter 2, a single Recipes table isn't such a hot idea. (Pun intended!) What about recipes that have ingredients other than meat and vegetables? What about the fact that some recipes have many ingredients and others have only a few? A correctly designed Recipes database will have a separate Recipe_Ingredients table with one row per recipe per ingredient. Each ingredient row will have only one ingredient, so no one row can be both beef and onions at the same time. You'll need to first find all the beef rows, then find all the onions rows, then difference them on RecipeID.

How about a more complex problem? Let's say you hate carrots, too. A set diagram to visualize the solution might look like Figure 7–4.

First you need to find the set of recipes that have beef, and then get the difference with either the set of recipes containing onions or the set containing

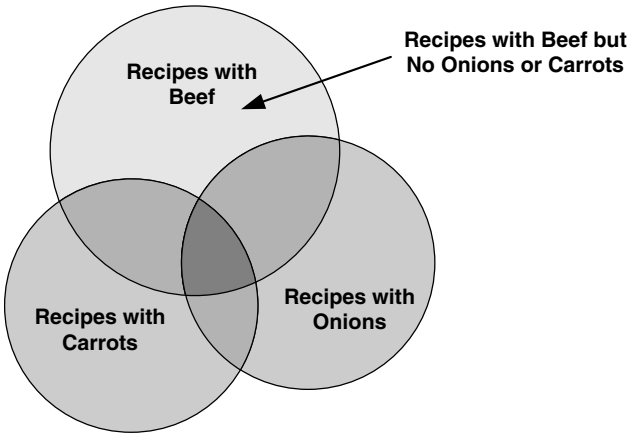


Figure 7–4 *Finding out which recipes have beef but no onions or carrots*

carrots. Take that result and get the difference again with the remaining set (onions or carrots) to leave only the recipes that have beef but no carrots or onions (the light-shaded area in the upper circle).

Problems You Can Solve with Difference

Unlike intersection (which looks for common members of two sets), difference looks for members that are in one set but *not* in another set. Here's just a small sample of the problems you can solve using a difference technique with data from the sample databases.

> *"Show me customers whose names are not the same as any employee."*
>
> *"Find all the customers who ordered a bicycle but did not order a helmet."*
>
> *"List the entertainers who played engagements for customer Bonnicksen but did not play any engagement for customer Rosales."*
>
> *"Show me the students who have an average score of 85 or better in Art but do not have an average score of 85 or better in Computer Science."*
>
> *"Find the bowlers who had a raw score of 155 or better at Thunderbird Lanes but not at Bolero Lanes."*
>
> *"Show me the recipes that have beef but not garlic."*

One of the limitations of using a pure difference is that the values must match in all the columns in each result set. This works well if you're finding the difference between two or more sets from the same table—for example, customers who ordered bicycles and customers who ordered helmets. It also

works well when you're finding the difference between sets from tables that have similar columns—for example, customer names and employee names.

In many cases, however, you'll want to find solutions that require a match on only a few column values from each set. For this type of problem, SQL provides an OUTER JOIN operation, which is an intersection on key values that includes the unmatched values from one or both of the two sets. Here's a sample of problems you can solve with an OUTER JOIN.

"Show me customers who do not live in the same city as any employees." (OUTER JOIN on the city name.)

"List customers and the entertainers they did not book." (OUTER JOIN on the engagement number.)

"Find the agents who are not in the same ZIP Code as any entertainer." (OUTER JOIN on the ZIP Code.)

"Show me the students who do not have the same first name as any teachers." (OUTER JOIN on the first name.)

"Find the bowlers who have an average of 150 or higher who have never bowled a game below 125." (OUTER JOIN on the bowler ID from two different tables.)

"Display all the ingredients for recipes that do not have carrots." (OUTER JOIN on the recipe ID.)

Don't worry! We'll show you all about solving these problems (and more) using OUTER JOINs in Chapter 9. Also, because few commercial implementations of SQL support EXCEPT (the keyword for difference), we'll show how to use an OUTER JOIN to solve many problems that might otherwise require an EXCEPT.

Union

So far we've discussed finding the items that are common in two sets (intersection) and the items that are different (difference). The third type of set operation involves adding two sets (union).

Union in Set Theory

Union lets you combine two sets of similar information into one set. As a scientist, you might be interested in combining two sets of chemical or physical sample data. For example, a pharmaceutical research chemist might have two

different sets of compounds that seem to provide a certain beneficial effect. The chemist can union the two sets to obtain a single list of all effective compounds.

Let's take a look at union in action by examining two sets of numbers. The first set of numbers is as follows.

1, 5, 8, 9, 32, 55, 78

The second set of numbers is as follows.

3, 7, 8, 22, 55, 71, 99

The union of these two sets of numbers is the numbers in both sets combined into one new set.

1, 5, 8, 9, 32, 55, 78, 3, 7, 22, 71, 99

Note that the values common to both sets, 8 and 55, appear only once in the answer. Also, the sequence of the numbers in the result set is not necessarily in any specific order. When you ask a database system to perform a UNION, the values returned won't necessarily be in sequence unless you explicitly include an ORDER BY clause. In SQL, you can also ask for a UNION ALL if you want to see the duplicate members.

The members of each set don't have to be just single values. In fact, you'll probably deal with sets of rows when working with SQL.

To find the union of two or more sets of complex members, all the members in each set you're trying to union must have the same number and type of attributes. For example, suppose you have a complex set like the one below. Each row represents a member of the set (a stew recipe), and each column denotes a particular attribute (an ingredient).

Potatoes	Water	Lamb	Peas
Rice	Chicken Stock	Chicken	Carrots
Pasta	Water	Tofu	Snap Peas
Potatoes	Beef Stock	Beef	Cabbage
Pasta	Water	Pork	Onions

A second set might look like the following.

Potatoes	Water	Lamb	Onions
Rice	Chicken Stock	Turkey	Carrots
Pasta	Vegetable Stock	Tofu	Snap Peas
Potatoes	Beef Stock	Beef	Cabbage
Beans	Water	Pork	Onions

The union of these two sets is the set of objects from both sets. Duplicates are eliminated.

Potatoes	Water	Lamb	Peas
Rice	Chicken Stock	Chicken	Carrots
Pasta	Water	Tofu	Snap Peas
Potatoes	Beef Stock	Beef	Cabbage
Pasta	Water	Pork	Onions
Potatoes	Water	Lamb	Onions
Rice	Chicken Stock	Turkey	Carrots
Pasta	Vegetable Stock	Tofu	Snap Peas
Beans	Water	Pork	Onions

Combining Result Sets Using a Union

It's a small leap from sets of complex objects to rows in SQL result sets. When you're dealing with rows in a set of data that you fetch with SQL, the attributes are the individual columns. For example, suppose you have a set of rows returned by a query like the following one. (These are recipes from John's cookbook.)

Recipe	Starch	Stock	Meat	Vegetable
Lamb Stew	Potatoes	Water	Lamb	Peas
Chicken Stew	Rice	Chicken Stock	Chicken	Carrots
Veggie Stew	Pasta	Water	Tofu	Snap Peas
Irish Stew	Potatoes	Beef Stock	Beef	Cabbage
Pork Stew	Pasta	Water	Pork	Onions

A second query result set might look like this one. (These are recipes from Mike's cookbook).

Recipe	Starch	Stock	Meat	Vegetable
Lamb Stew	Potatoes	Water	Lamb	Peas
Turkey Stew	Rice	Chicken Stock	Turkey	Carrots
Veggie Stew	Pasta	Vegetable Stock	Tofu	Snap Peas
Irish Stew	Potatoes	Beef Stock	Beef	Cabbage
Pork Stew	Beans	Water	Pork	Onions

The union of these two sets is all the rows in both sets. Maybe John and Mike decided to write a cookbook together, too!

Recipe	Starch	Stock	Meat	Vegetable
Lamb Stew	Potatoes	Water	Lamb	Peas
Chicken Stew	Rice	Chicken Stock	Chicken	Carrots
Veggie Stew	Pasta	Water	Tofu	Snap Peas
Irish Stew	Potatoes	Beef Stock	Beef	Cabbage
Pork Stew	Pasta	Water	Pork	Onions
Turkey Stew	Rice	Chicken Stock	Turkey	Carrots
Veggie Stew	Pasta	Vegetable Stock	Tofu	Snap Peas
Pork Stew	Beans	Water	Pork	Onions

Let's assume you have a nice database containing all your favorite recipes. You really like recipes with either beef or onions, so you want a list of recipes that contain either ingredient. Figure 7–5 (on page 232) shows you the set diagram that helps you visualize how to solve this problem.

The upper circle represents the set of recipes that contain beef. The lower circle represents the set of recipes that contain onions. The union of the two circles gives you all the recipes that contain either ingredient, with duplicates eliminated where the two sets overlap. As you probably know, you first ask SQL to fetch all the recipes that have beef. In the second query, you ask SQL

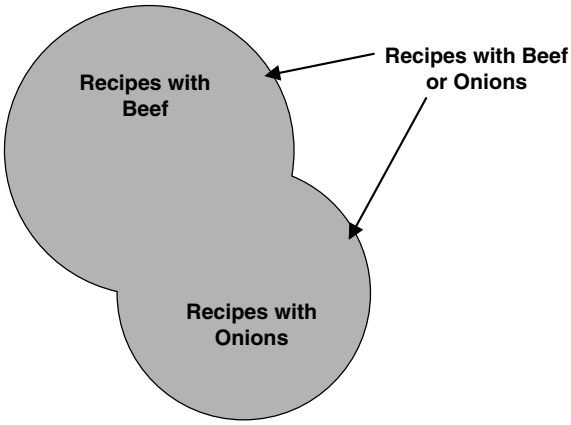


Figure 7–5 *Finding out which recipes have either beef or onions*

to fetch all the recipes that have onions. As you'll see later, the SQL keyword UNION links the two queries to get the final answer.

By now you know that it's not a good idea to design a recipes database with a single table. Instead, a correctly designed recipes database will have a separate Recipe_Ingredients table with one row per recipe per ingredient. Each ingredient row will have only one ingredient, so no one row can be both beef or onions at the same time. You'll need to first find all the recipes that have a beef row, then find all the recipes that have an onions row, and then union them.

Problems You Can Solve with Union

A union lets you "mush together" rows from two similar sets—with the added advantage of no duplicate rows. Here's a sample of the problems you can solve using a union technique with data from the sample databases.

"Show me all the customer and employee names and addresses."

"List all the customers who ordered a bicycle combined with all the customers who ordered a helmet."

"List the entertainers who played engagements for customer Bonnicksen combined with all the entertainers who played engagements for customer Rosales."

"Show me the students who have an average score of 85 or better in Art together with the students who have an average score of 85 or better in Computer Science."

"Find the bowlers who had a raw score of 155 or better at Thunderbird Lanes combined with bowlers who had a raw score of 140 or better at Bolero Lanes."

"Show me the recipes that have beef together with the recipes that have garlic."

As with other "pure" set operations, one of the limitations is that the values must match in all the columns in each result set. This works well if you're unioning two or more sets from the same table—for example, customers who ordered bicycles and customers who ordered helmets. It also works well when you're performing a union on sets from tables that have like columns—for example, customer names and addresses and employee names and addresses. We'll explore the uses of the SQL UNION operator in detail in Chapter 10.

In many cases where you would otherwise union rows from the same table, you'll find that using DISTINCT (to eliminate the duplicate rows) with complex criteria on joined tables will serve as well. We'll show you all about solving problems this way using JOINs in Chapter 8, INNER JOINs.

SQL Set Operations

Now that you have a basic understanding of set operations, let's look briefly at how they're implemented in SQL.

Classic Set Operations versus SQL

As noted earlier, not many commercial database systems yet support set intersection (INTERSECT) or set difference (EXCEPT) directly. The current SQL Standard, however, clearly defines how these operations should be implemented. We think that these set operations are important enough to at least warrant an overview of the syntax.

As promised, we'll show you alternative ways to solve an intersection or difference problem in later chapters using JOINs. Because most database systems do support UNION, Chapter 10 is devoted to its use. The remainder of this chapter gives you an overview of all three operations.

Finding Common Values: INTERSECT

Let's say you're trying to solve the following seemingly simple problem.

"Show me the orders that contain both a bike and a helmet."

Translation	Select the distinct order numbers from the order details table where the product number is in the list of bike and helmet product numbers
Clean Up	Select ~~the~~ distinct order numbers from ~~the~~ order details ~~table~~ where ~~the~~ product number ~~is~~ in ~~the list of~~ bike and helmet product numbers

SQL	`SELECT DISTINCT OrderNumber`
	`FROM Order_Details`
	`WHERE ProductNumber IN (1, 2, 6, 10, 11, 25, 26)`

❖ **Note** Readers familiar with SQL might ask why we didn't JOIN Order_Details to Products and look for bike or helmet product names. The simple answer is that we haven't introduced the concept of a JOIN yet, so we built this example on a single table using IN and a list of known bike and helmet product numbers.

That seems to do the trick at first, but the answer includes orders that contain either a bike *or* a helmet, and you really want to find ones that contain *both* a bike *and* a helmet! If you visualize orders with bicycles and orders with helmets as two distinct sets, it's easier to understand the problem. Figure 7–6 shows one possible relationship between the two sets of orders using a set diagram.

Actually, there's no way to predict in advance what the relationship between two sets of data might be. In Figure 7–6, some orders have a bicycle in the list of products ordered, but no helmet. Some have a helmet, but no bicycle. The overlapping area, or intersection, of the two sets is where you'll find orders that have both a bicycle and a helmet. Figure 7–7 shows another case where *all* orders that contain a helmet also contain a bicycle, but some orders that contain a bicycle do not contain a helmet.

Seeing "both" in your request suggests you're probably going to have to break the solution into separate sets of data and then link the two sets in some way. (Your request also needs to be broken into two parts.)

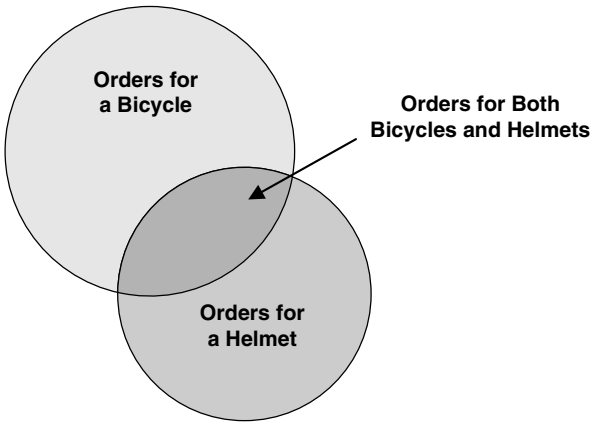


Figure 7–6 *One possible relationship between two sets of orders*

"Show me the orders that contain a bike."

Translation	Select the distinct order numbers from the order details table where the product number is in the list of bike product numbers
Clean Up	Select ~~the~~ distinct order numbers from ~~the~~ order details ~~table~~ where ~~the~~ product number ~~is~~ in ~~the list of~~ bike product numbers

SQL	`SELECT DISTINCT OrderNumber`
	`FROM Order_Details`
	`WHERE ProductNumber IN (1, 2, 6, 11)`

"Show me the orders that contain a helmet."

Translation	Select the distinct order numbers from the order details table where the product number is in the list of helmet product numbers

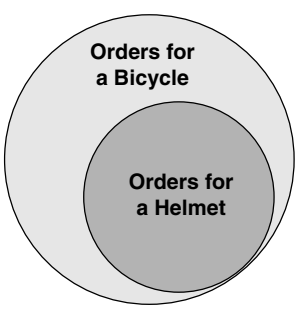


Figure 7–7 *All orders for a helmet also contain an order for a bicycle.*

Clean Up	Select ~~the~~ distinct order numbers from ~~the~~ order details ~~table~~ where ~~the~~ product number ~~is~~ in ~~the list of~~ helmet product numbers

SQL	`SELECT DISTINCT OrderNumber`
	`FROM Order_Details`
	`WHERE ProductNumber IN (10, 25, 26)`

Now you're ready to get the final solution by using—you guessed it—an *intersection* of the two sets. Figure 7–8 shows the SQL syntax diagram that handles this problem. (Note that you can use INTERSECT more than once to combine multiple SELECT statements.)

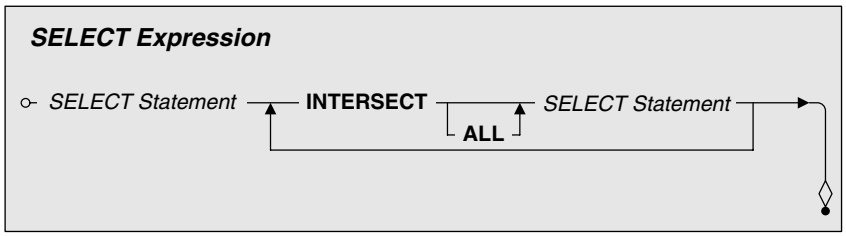


Figure 7–8 *Linking two SELECT statements with INTERSECT*

You can now take the two parts of your request and link them with an INTERSECT operator to get the correct answer.

```
SQL         SELECT DISTINCT OrderNumber
            FROM Order_Details
            WHERE ProductNumber IN (1, 2, 6, 11)
            INTERSECT
            SELECT DISTINCT OrderNumber
            FROM Order_Details
            WHERE ProductNumber IN (10, 25, 26)
```

The sad news is that not many commercial implementations of SQL yet support the INTERSECT operator. But all is not lost! Remember that the primary key of a table uniquely identifies each row. (You don't have to match on all the fields in a row—just the primary key—to find unique rows that intersect.) We'll show you an alternative method (JOIN) in Chapter 8 that can solve this type of problem in another way. The good news is that most commercial implementations of SQL *do* support JOIN.

Finding Missing Values: EXCEPT (DIFFERENCE)

Okay, let's go back to the bicycles and helmets problem again. Let's say you're trying to solve this seemingly simple request as follows.

"Show me the orders that contain a bike but not a helmet."

Translation Select the distinct order numbers from the order details table where the product number is in the list of bike product numbers and product number is not in the list of helmet product numbers

Clean Up Select ~~the~~ distinct order numbers from ~~the~~ order details ~~table~~ where ~~the~~ product number ~~is~~ in ~~the list of~~ bike product numbers and product number ~~is~~ not in ~~the list of~~ helmet product numbers

SQL
```
SELECT DISTINCT OrderNumber
FROM Order_Details
WHERE ProductNumber IN (1, 2, 6, 11)
   AND ProductNumber NOT IN (10, 25, 26)
```

Unfortunately, the answer shows you orders that contain only a bike! The problem is that the first IN clause finds detail rows containing a bicycle, but the second IN clause simply eliminates helmet rows. If you visualize orders with bicycles and orders with helmets as two distinct sets, you'll find this easier to understand. Figure 7–9 shows one possible relationship between the two sets of orders.

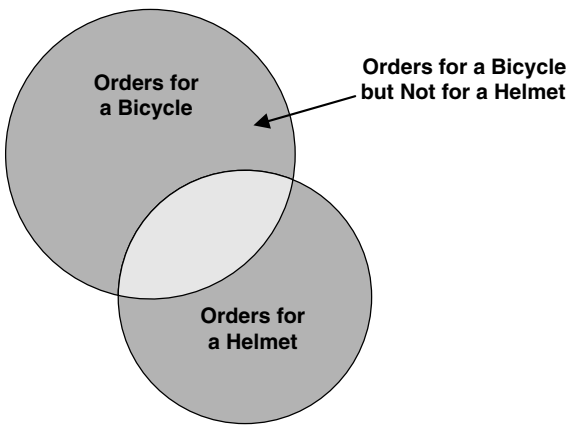


Figure 7–9 *Orders for a bicycle that do not also contain a helmet*

Seeing "except" or "but not" in your request suggests you're probably going to have to break the solution into separate sets of data and then link the two sets in some way. (Your request also needs to be broken into two parts.)

"Show me the orders that contain a bike."

Translation Select the distinct order numbers from the order details table where the product number is in the list of bike product numbers

Clean Up Select ~~the~~ distinct order numbers from ~~the~~ order details ~~table~~ where ~~the~~ product number ~~is~~ in ~~the list of~~ bike product numbers

```
SQL         SELECT DISTINCT OrderNumber
            FROM Order_Details
            WHERE ProductNumber IN (1, 2, 6, 11)
```

"Show me the orders that contain a helmet."

Translation Select the distinct order numbers from the order details table where the product number is in the list of helmet product numbers

Clean Up Select ~~the~~ distinct order numbers from ~~the~~ order details ~~table~~ where ~~the~~ product number ~~is~~ in ~~the list of~~ helmet product numbers

```
SQL         SELECT DISTINCT OrderNumber
            FROM Order_Details
            WHERE ProductNumber IN (10, 25, 26)
```

Now you're ready to get the final solution by using—you guessed it—a *difference* of the two sets. SQL uses the EXCEPT keyword to denote a difference operation. Figure 7–10 shows you the SQL syntax diagram that handles this problem.

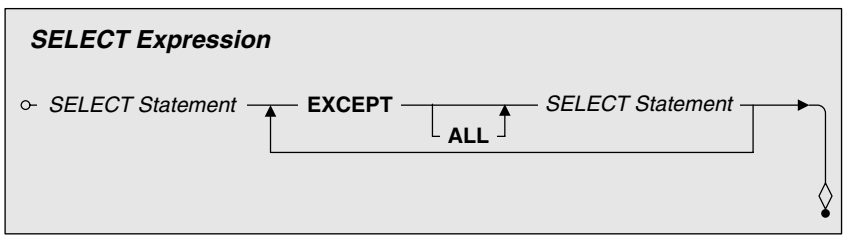


Figure 7–10 *Linking two SELECT statements with EXCEPT*

You can now take the two parts of your request and link them with an EXCEPT operator to get the correct answer.

```
SQL         SELECT DISTINCT OrderNumber
            FROM Order_Details
            WHERE ProductNumber IN (1, 2, 6, 11)
            EXCEPT
            SELECT DISTINCT OrderNumber
            FROM Order_Details
            WHERE ProductNumber IN (10, 25, 26)
```

Remember from our earlier discussion about the difference operation that the sequence of the sets matters. In this case you're asking for bikes "except" helmets. If you want to find out the opposite case—orders for helmets that do not include bikes—you can turn it around as follows.

SQL
```
SELECT DISTINCT OrderNumber
FROM Order_Details
WHERE ProductNumber IN (10, 25, 26)
EXCEPT
SELECT DISTINCT OrderNumber
FROM Order_Details
WHERE ProductNumber IN (1, 2, 6, 11)
```

The sad news is that not many commercial implementations of SQL yet support the EXCEPT operator. Hang on to your helmet! Remember that the primary key of a table uniquely identifies each row. (You don't have to match on all the fields in a row—just the primary key—to find unique rows that are different.) We'll show you an alternative method (OUTER JOIN) in Chapter 9 that can solve this type of problem in another way. The good news is that most commercial implementations of SQL *do* support OUTER JOIN.

Combining Sets: UNION

One more problem about bicycles and helmets, then we'll pedal on to the next chapter. Let's say you're trying to solve this request, which looks simple enough on the surface.

"Show me the orders that contain either a bike or a helmet."

Translation: Select the distinct order numbers from the order details table where the product number is in the list of bike and helmet product numbers

Clean Up: Select ~~the~~ distinct order numbers from ~~the~~ order details ~~table~~ where ~~the~~ product number ~~is~~ in ~~the list of~~ bike and helmet product numbers

SQL
```
SELECT DISTINCT OrderNumber
FROM Order_Details
WHERE ProductNumber IN (1, 2, 6, 10, 11, 25, 26)
```

Actually, that works just fine! So why use a UNION to solve this problem? The truth is, you probably would not. However, if we make the problem more complicated, a UNION would be useful.

"List the customers who ordered a bicycle together with the vendors who provide bicycles."

Unfortunately, answering this request involves creating a couple of queries using JOIN operations, then using UNION to get the final result. Because we haven't shown you how to do a JOIN yet, we'll save solving this problem for Chapter 10. Gives you something to look forward to, doesn't it?

Let's get back to the "bicycles or helmets" problem and solve it with a UNION. If you visualize orders with bicycles and orders with helmets as two distinct sets, then you'll find it easier to understand the problem. Figure 7–11 shows you one possible relationship between the two sets of orders.

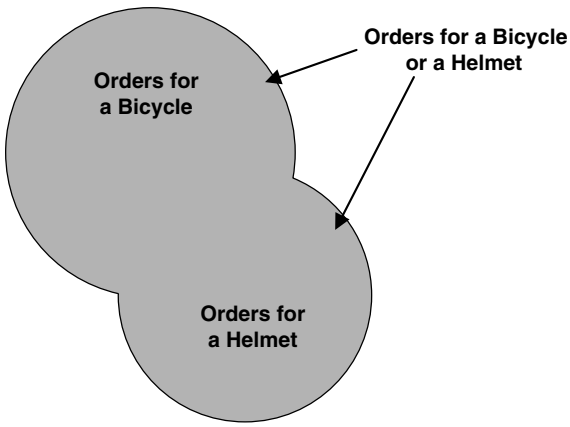


Figure 7–11 *Orders for bicycles or helmets*

Seeing "either," "or," or "together" in your request suggests that you'll need to break the solution into separate sets of data and then link the two sets with a UNION. This particular request can be broken into two parts.

"Show me the orders that contain a bike."

Translation	Select the distinct order numbers from the order details table where the product number is in the list of bike product numbers
Clean Up	Select ~~the~~ distinct order numbers from ~~the~~ order details ~~table~~ where ~~the~~ product number ~~is~~ in ~~the list of~~ bike product numbers

SQL	`SELECT DISTINCT OrderNumber`
	`FROM Order_Details`
	`WHERE ProductNumber IN (1, 2, 6, 11)`

"Show me the orders that contain a helmet."

Translation: Select the distinct order numbers from the order details table where the product number is in the list of helmet product numbers

Clean Up: Select ~~the~~ distinct order numbers from ~~the~~ order details ~~table~~ where ~~the~~ product number ~~is~~ in ~~the list of~~ helmet product numbers

SQL:
```
SELECT DISTINCT OrderNumber
FROM Order_Details
WHERE ProductNumber IN (10, 25, 26)
```

Now you're ready to get the final solution by using—you guessed it—a *union* of the two sets. Figure 7–12 shows the SQL syntax diagram that handles this problem.

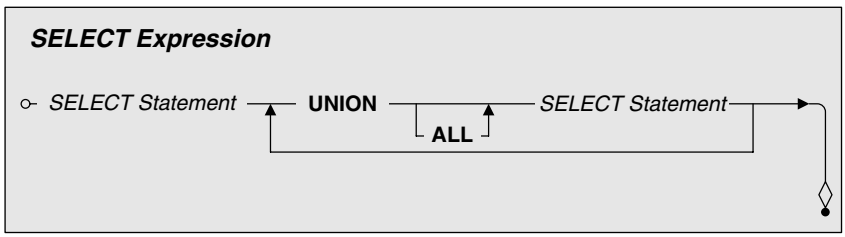


Figure 7–12 *Linking two SELECT statements with UNION*

You can now take the two parts of your request and link them with a UNION operator to get the correct answer.

SQL:
```
SELECT DISTINCT OrderNumber
FROM Order_Details
WHERE ProductNumber IN (1, 2, 6, 11)
UNION
SELECT DISTINCT OrderNumber
FROM Order_Details
WHERE ProductNumber IN (10, 25, 26)
```

The good news is that most commercial implementations of SQL support the UNION operator. As is perhaps obvious from the examples, a UNION might be doing it the hard way when you want to get an "either-or" result from a single table. UNION is most useful for compiling a list from several similarly structured but different tables. We'll explore UNION in much more detail in Chapter 10.

SUMMARY

We began this chapter by discussing the concept of a set. Next, we discussed each of the major set operations implemented in SQL in detail—intersection, difference, and union. We showed how to use set diagrams to visualize the problem you're trying to solve. Finally, we introduced you to the basic SQL syntax and keywords (INTERSECT, EXCEPT, and UNION) for all three operations just to whet your appetite.

At this point you're probably saying, "Wait a minute, why did you show me three kinds of set operations—two of which I probably can't use?" Remember the title of the chapter: Thinking in Sets. If you're going to be at all successful solving complex problems, you'll need to break your problem into result sets of information that you then link back together.

So, if your problem involves "it must be this *and* it must be that," you might need to solve the "this" and then the "that" and then link them to get your final solution. The SQL Standard defines a handy INTERSECT operation—but an INNER JOIN might work just as well. Read on in Chapter 8.

Likewise, if your problem involves "it must be this *but it must not be* that," you might need to solve the "this" and then the "that" and then subtract the "that" from the "this" to get your answer. We showed you the SQL Standard EXCEPT operation, but an OUTER JOIN might also do the trick. Get the details in Chapter 9.

Finally, we showed you how to add sets of information using a UNION. As promised, we'll really get into UNION in Chapter 10.

8

INNER JOINs

"Do not quench your inspiration and your imagination;
do not become the slave of your model."
—Vincent van Gogh

Topics Covered in This Chapter

What Is a JOIN?

The INNER JOIN

Uses for INNER JOINs

Sample Statements

Summary

Problems for You to Solve

Up to this point, we have primarily focused on solving problems using single tables. You now know how to get simple answers from one table. You also know how to get slightly more complex answers by using expressions or by sorting the result set. In other words, you now can draw the perfect eyes, chin, mouth, or nose. In this chapter, we'll show you how to link or join multiple parts to form a portrait.

What Is a JOIN?

In Chapter 2, Ensuring Your Database Structure Is Sound, we emphasized the importance of separating the data in your tables into individual subjects. Most problems you need to solve in real life, however, require that you link data from multiple tables—customers and their orders, customers and the entertainers they booked, bowlers and their scores, students and the classes they took, or recipes and the ingredients you need. To solve these more complex

problems, you must link, or join, multiple tables to find your answer. You use the *JOIN* keyword to do so.

The previous chapter showed how useful it is to intersect two sets of data to solve problems. As you recall, however, an INTERSECT involves matching all the columns in both result sets to get the answer. A JOIN is also an intersection, but it's different because you ask your database system to perform a JOIN only on the columns you specify. Thus, a JOIN lets you intersect two very dissimilar tables on matching column values. For example, you can use a JOIN to link customers to their orders by matching the CustomerID in the Customers table to the CustomerID in the Orders table.

As you'll see later, you specify a JOIN as part of the FROM clause in an SQL statement. A JOIN defines a "logical table" that is the result of linking two tables or result sets. By placing the JOIN in a FROM clause, you define a linking of tables *from* which the query extracts the final result set. In other words, the JOIN replaces the single table name you learned to use in the FROM clause in earlier chapters. As you'll learn later in this chapter, you can also specify multiple JOIN operations to create a complex result set on more than two tables.

The INNER JOIN

The SQL Standard defines several ways to perform a JOIN, the most common of which is the *INNER JOIN*. Imagine for a moment that you're linking students and the classes for which they registered. You might have some students who have been accepted to attend the school but have not yet registered for any classes, and you might also have some classes that are on the schedule but do not yet have any students registered.

An INNER JOIN between the Students table and the Classes table returns rows in the Students table linked with the related rows in the Classes table (via the Student_Schedules table)—but it returns neither students who have not yet registered for any classes nor any classes for which no student is registered. An INNER JOIN returns only those rows where the linking values match in both of the tables or in result sets.

What's "Legal" to JOIN?

Most of the time, you specify the primary key from one table and the related foreign key from the second table as the link that JOIN uses. If you remember

from Chapter 2, a foreign key must be the same data type as its related primary key. However, it's also "legal" to JOIN two tables or result sets on any columns that have what the SQL Standard calls "JOIN eligible" data types.

In general, you can join a character column to another character column or expression, any type of number column (for example, an integer) to any other type of number column (perhaps a floating-point value), and any date column to another date column. This allows you, for example, to JOIN rows from the Customers table to rows from the Employees table on the city or ZIP Code columns (perhaps to find out which Customers and Employees live in the same city or postal region).

❖ **Note** Just because you ***can*** define a JOIN on any JOIN eligible columns in two tables doesn't mean you ***should***. The linking columns must have the same data meaning for the JOIN to make sense. For example, it doesn't make sense to JOIN customer name with employee address even though both columns are character data type. You won't get any rows in the result set unless someone has put a name in the employee address column by mistake. Likewise, it doesn't make sense to JOIN StudentID with ClassID even though both are numbers. You might get some rows in the result set, but they won't make any sense.

Even when it makes sense to JOIN linking columns, you might end up constructing a request that takes a long time to solve. For example, if you ask for a JOIN on columns for which your database administrator has not defined an index, your database system might have to do a lot of extra work. Also, if you ask for a JOIN on expressions—for example, a concatenation of first name and last name from two tables—your database system must not only form the result column from your expression for all rows but also might have to perform multiple scans of all the data in both tables to return the correct result.

Column References

Before we jump into the syntax for a JOIN, there's a key bit of information that we haven't covered yet. Because you've been creating queries on a single table, you haven't had to worry about qualifying column names. But when you start to build queries that include multiple tables (as you will when you

use a JOIN), you'll often include two or more tables that each have columns with the same name. If you remember from Chapter 2, we recommended that you create a foreign key in a related table by copying the primary key—including its name—from one table into another.

So, how do you make it crystal clear to your database system which copy of a field you are talking about in your query syntax? The simple answer is that you provide a *column reference* that includes the table name. Figure 8–1 shows the diagram for a column reference.

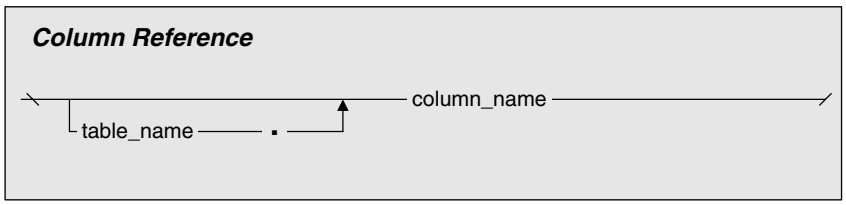


Figure 8–1 *The syntax diagram of a column reference*

Although you can use only the column name by itself in any clause in a statement that you write in SQL, you can also explicitly qualify a column name with the name of its parent table. If the column name isn't unique in all the tables you include in your FROM clause, then you *must* qualify the column name with the name of its parent table. Here's how you would write a simple SELECT statement on the Employees table to incorporate qualified column names.

```
SQL         SELECT Employees.FirstName, Employees.LastName,
               Employees.PhoneNumber
            FROM Employees
```

Now that we've covered that little tidbit, you can move on to studying the syntax of a JOIN operation.

Syntax

You can think of what you've studied so far as taking a nice ride down a country lane or a quick jaunt across town to pick up some groceries. Now let's strap on our seat belts and venture out onto the highway—let's examine the INNER JOIN syntax.

Using Tables

We'll start with something simple—an INNER JOIN on two tables. Figure 8–2 shows the syntax for creating the query.

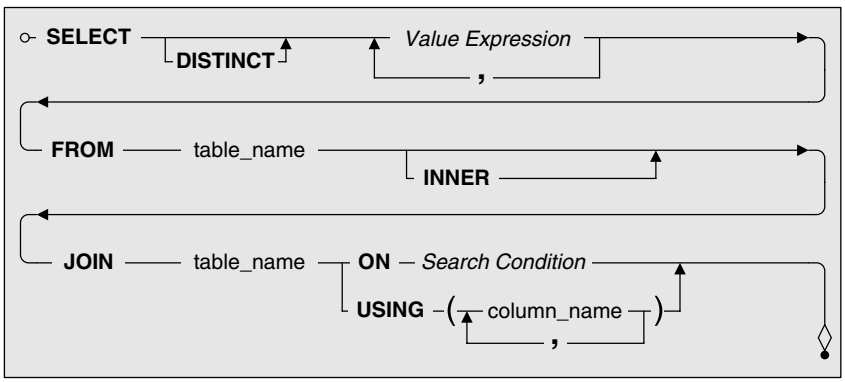


Figure 8–2 *The syntax diagram of a query using an INNER JOIN on two tables*

As you can see, the FROM clause is now just a little more complicated. (We left out the WHERE and ORDER BY clauses for now to simplify things.) Instead of a single table name, you specify two table names and link them with the JOIN keyword. Note that the INNER keyword, which is optional, specifies the type of JOIN. As you'll learn in the next chapter, you can also specify an OUTER JOIN. If you don't explicitly state the type of JOIN you want, the default is INNER. We recommend that you always explicitly state the type of JOIN you want so that the nature of your request is clear.

❖ **Note** Those who are following along with the complete syntax diagrams in Appendix A, SQL Standard Diagrams, will find *`Table Reference`* `JOIN` *`Table Reference`* described as part of the *Joined Table* defined term. *Table Reference* can be either a `table_name` or a *`Joined Table`*, and the FROM clause of a SELECT statement uses *`Table Reference`*. We "rolled up" these complex definitions into a single diagram to make it easy to study a simple two-table JOIN. We'll be using this same simplification technique in diagrams throughout the remainder of this chapter.

The critical part of an INNER JOIN is the ON or USING clause that follows the second table and tells your database system how to perform the JOIN. To

solve the JOIN, your database system logically combines every row in the first table with every row in the second table. (This combination of all rows from one table with all rows from a second table is called a *Cartesian product*.) It then applies the criteria in the ON or USING clauses to filter out the actual rows to be returned.

You learned about using a *search condition* to form a WHERE clause in Chapter 6, Filtering Your Data. You can use a search condition in the ON clause within a JOIN to specify a logical test that must be true in order to return any two linked rows. Keep in mind that it only makes sense to write a search condition that compares at least one column from the first table with at least one column from the second table. Although you can write a very complex search condition, you'll typically specify a simple equals comparison test on the primary key columns from one table with the foreign key columns from the other table.

Let's look at a simple example. In a well-designed database, you should break out complex classification names into a second table and then link the names back to the primary subject table via a simple key value. You do this to help prevent data entry errors. Anyone using your database chooses from a list of classification names rather than typing the name (and perhaps misspelling it) in each row. For example, in the Recipes sample database, recipe classes appear in a table separate from recipes. Figure 8–3 shows the relationship between the Recipe_Classes and Recipes tables.

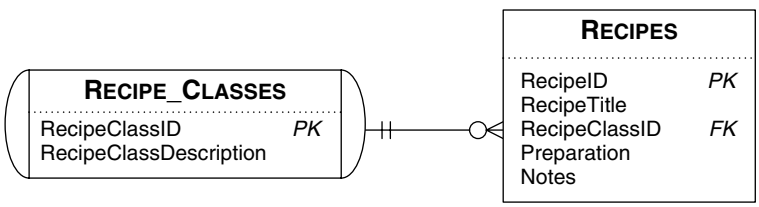


Figure 8–3 *Recipe class descriptions are in a table separate from the Recipes table.*

When you want to retrieve information about recipes and the related RecipeClassDescription from the database, you don't want to see the RecipeClassID code numbers from the Recipes table. Let's see how to approach this problem with a JOIN.

❖ **Note** Throughout this chapter, we use the "Request/Translation/Clean Up/SQL" technique introduced in Chapter 4, Creating a Simple Query.

"Show me the recipe title, preparation, and recipe class description of all recipes in my database."

Translation	Select recipe title, preparation, and recipe class description from the recipe classes table joined with the recipes table on recipe class ID in the recipe classes table matching recipe class ID in the recipes table
Clean Up	Select recipe title, preparation, ~~and~~ recipe class description from ~~the~~ recipe classes ~~table~~ inner join~~ed with the~~ recipes ~~table~~ on recipe_classes.recipe class ID ~~in the recipe classes table matching~~ = recipes.recipe class ID ~~in the recipes table~~

SQL

```
SELECT RecipeTitle, Preparation,
   RecipeClassDescription
FROM Recipe_Classes
INNER JOIN Recipes
ON Recipe_Classes.RecipeClassID =
   Recipes.RecipeClassID
```

❖ **Note** You might have noticed that we've started to format the Clean Up step into phrases that more closely mirror the final set of clauses we need in SQL. As you begin to build more complex queries, we recommend this technique to help you move from the Clean Up step to the final SQL.

When beginning to use multiple tables in your FROM clause, you should always fully qualify each column name with the table name, wherever you use it, to make absolutely clear what column from what table you want. (Now you know why we took a minute to explain a column reference!) Note that we *had to* qualify the name of RecipeClassID in the ON clause because there are two columns named RecipeClassID—one in the Recipes table and one in the Recipe_Classes table. We didn't have to qualify RecipeTitle, Preparation, or RecipeClassDescription in the SELECT clause because each of these column names appears only once in all the tables. If we want to include RecipeClassID in the output, we must tell the database system *which* RecipeClassID we want—the one from Recipe_Classes or the one from Recipes. To write the query with all the names fully qualified, we should say this:

```
SQL         SELECT Recipes.RecipeTitle,
               Recipes.Preparation,
               Recipe_Classes.RecipeClassDescription
            FROM Recipe_Classes
            INNER JOIN Recipes
            ON Recipe_Classes.RecipeClassID =
               Recipes.RecipeClassID
```

❖ **Note** Although most commercial implementations of SQL support the JOIN keyword, some do not. If your database does not support JOIN, you can still solve the previous problem by listing all the tables you need in the FROM clause and then moving your search condition from the ON clause to the WHERE clause. In databases that do not support JOIN, you solve our example problem like this:

```
SELECT Recipes.RecipeTitle, Recipes.Preparation,
   Recipe_Classes.RecipeClassDescription
FROM Recipe_Classes, Recipes
WHERE Recipe_Classes.RecipeClassID =
   Recipes.RecipeClassID
```

For a beginner, this syntax is probably much more intuitive for simple queries. However, the SQL Standard syntax allows you to fully define the source for the final result set entirely within the FROM clause. Think of the FROM clause as fully defining a linked result set from which the database system obtains your answer. In the SQL Standard, you use the WHERE clause only to filter rows out of the result set defined by the FROM clause.

Not too difficult, is it? But what happened to the USING clause that we showed you in Figure 8–2? If the matching columns in the two tables have the same name and all you want to do is join on equal values, use the USING clause and list the column names. Let's do the previous problem again with USING.

"Show me the recipe title, preparation, and recipe class description of all recipes in my database."

Translation	Select recipe title, preparation, and recipe class description from the recipe classes table joined with the recipes table using recipe class ID
Clean Up	Select recipe title, preparation, ~~and~~ recipe class description from ~~the~~ recipe classes ~~table~~ join~~ed with the~~ recipes ~~table~~ using recipe class ID

```
SQL         SELECT Recipes.RecipeTitle, Recipes.Preparation,
               Recipe_Classes.RecipeClassDescription
            FROM Recipe_Classes
            INNER JOIN Recipes
            USING (RecipeClassID)
```

Some database systems do not yet support USING. If you find that you can't use USING with your database, you can always get the same result with an ON clause and an equals comparison.

❖ **Note** The SQL Standard also defines a NATURAL JOIN, which links the two specified tables by matching all the columns with the same name. If the only common columns are the linking columns and your database supports NATURAL JOIN, you can solve our example problem like this:

```
SELECT Recipes.RecipeTitle, Recipes.Preparation,
   Recipe_Classes.RecipeClassDescription
FROM Recipe_Classes
NATURAL INNER JOIN Recipes
```

Do not specify an ON or USING clause when using the NATURAL keyword. Keep in mind that the INNER keyword is optional. If you specify NATURAL JOIN, an INNER join is assumed.

As mentioned earlier in this section, your database system logically creates the combination of every row in the first table with every row in the second table and then applies the criteria you specify in ON or USING. This sounds like a lot of extra work for your database to first build all the combinations and then filter out the potentially few matching rows.

Rest assured that all modern relational database systems evaluate the entire JOIN clause before starting to fetch rows. In the example we have been using so far, many database systems begin to solve this request by first fetching a row from Recipe_Classes. The database then uses an internal link—an index (if one has been defined by the designer of the tables)—to quickly find any matching rows in the Recipes table for the first row in the Recipe_Classes table before moving on to the next row in Recipe_Classes. In other words, your database uses a smart or optimized plan to fetch only the rows that match. This won't seem important when your database tables contain only a few hundred rows, but it makes a big difference when your database has to deal with hundreds of thousands of rows!

Assigning Correlation (Alias) Names to Tables

The SQL Standard defines a way to assign an alias name—known as a *correlation name* in the Standard—to any table you list in your FROM clause. This feature can be very handy for building complex queries using tables that have long, descriptive names. You can assign a short correlation name to a table to make it easier to explicitly reference columns in a table with a long name.

Figure 8–4 shows how to assign a correlation name to a table in a FROM clause.

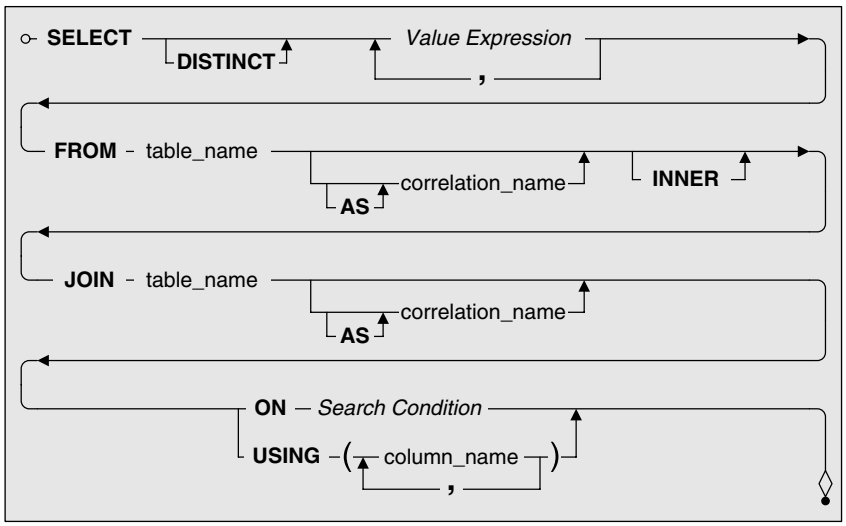


Figure 8–4 *Assigning a correlation (alias) name to a table in a FROM clause*

To assign a correlation name to a table, follow the table name with the optional keyword AS and then the correlation name you want to assign. (As with all optional keywords, we recommend including AS in order to make the query easier to read and understand.) After you have assigned a correlation name to a table, you use that name in place of the original table name in all other clauses, including the SELECT clause, the search conditions in the ON and WHERE clauses, and the ORDER BY clause. This can be confusing because you tend to write the SELECT clause before you write the FROM clause. If you plan to give a table an alias in the FROM clause, you must use that alias when you qualify column names in the SELECT clause.

Let's reformulate the sample query we've been using with correlation names just to see how it looks. The query using R as the correlation name for the Recipes table and RC as the correlation name for the Recipe_Classes table is shown here.

```
SQL         SELECT R.RecipeTitle, R.Preparation,
               RC.RecipeClassDescription
            FROM Recipe_Classes AS RC
            INNER JOIN Recipes AS R
            ON RC.RecipeClassID = R.RecipeClassID
```

Suppose you want to add a filter to see only recipes of class Main course or Dessert. (See Chapter 6 for details about defining filters.) After you assign a correlation name, you must continue to use the new name in all references to the table. Here's the SQL.

```
SQL         SELECT R.RecipeTitle, R.Preparation,
               RC.RecipeClassDescription
            FROM Recipe_Classes AS RC
            INNER JOIN Recipes AS R
            ON RC.RecipeClassID = R.RecipeClassID
            WHERE RC.RecipeClassDescription = 'Main course'
            OR RC.RecipeClassDescription = 'Dessert'
```

You don't have to assign a correlation name to all tables. In the previous example, we could have assigned a correlation name only to Recipes or only to Recipe_Classes.

In some cases, you *must* assign a correlation name to a table in a complex JOIN. Let's hop over to the Bowling League database to examine a case where this is true. Figure 8–5 shows you the relationship between the Teams and Bowlers tables.

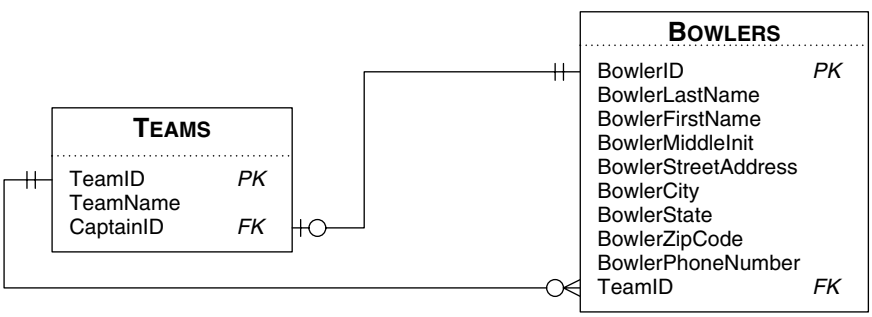


Figure 8–5 *The relationships between Teams and Bowlers*

As you can see, TeamID is a foreign key in the Bowlers table that lets you find the information for all bowlers on a team. One of the bowlers on a team is the team captain, so there's also a link from BowlerID in the Bowlers table to CaptainID in the Teams table.

If you want to list the team name, the name of the team captain, and the names of all the bowlers in one request, you must include *two* copies of the Bowlers table in your query—one to link to CaptainID to retrieve the name of the team captain and another to link to TeamID to get a list of all the team members. In this case, you *must* assign an alias name to one or both copies of the Bowlers table so that your database system can differentiate between the copy that links in the captain's name and the copy that provides the list of all team members. Later in this chapter, we'll show an example that requires including multiple copies of one table and assigning alias names. You can find this example using the Bowling League database in the More Than Two Tables subsection of Sample Statements.

Embedding a SELECT Statement

Let's make it more interesting. In most implementations of SQL, you can substitute an entire SELECT statement for any table name in your FROM clause. In the SQL Standard, an embedded SELECT statement like this is called a *derived table*. If you think about it, using a SELECT statement is simply a way to derive a subset of data from one or more tables. Of course, you must assign a correlation name so that the result of evaluating your embedded query has a name. Figure 8–6 shows how to assemble a JOIN clause using embedded SELECT statements.

Notice in the figure that a SELECT statement can include all query clauses except an ORDER BY clause. Also, you can mix and match SELECT statements with table names on either side of the INNER JOIN keywords.

Let's look at the Recipes and Recipe_Classes tables again. We'll assume that your request still needs only main courses and desserts. Here's the query again with the Recipe_Classes table filtered in a SELECT statement that's part of the INNER JOIN.

```
SQL         SELECT R.RecipeTitle, R.Preparation,
               RCFiltered.ClassName
            FROM (SELECT RecipeClassID,
                  RecipeClassDescription AS ClassName
                  FROM Recipe_Classes AS RC
```

```
        WHERE RC.ClassName = 'Main course' OR
        RC.ClassName = 'Dessert') AS RCFiltered
INNER JOIN Recipes AS R
ON RCFiltered.RecipeClassID = R.RecipeClassID
```

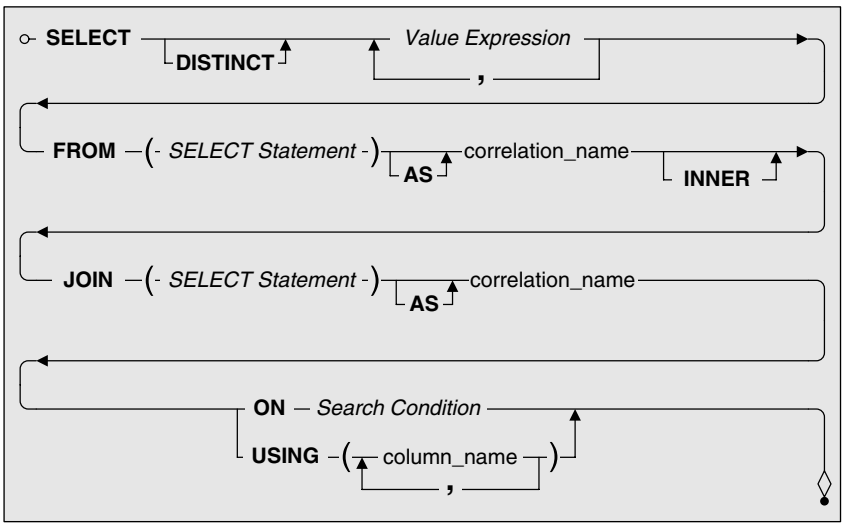


Figure 8–6 *Replacing table names with SELECT statements in a JOIN*

❖ **Note** Some database systems do not support embedding a SELECT statement inside a FROM clause. If your system does not support this feature, you can often save the inner SELECT statement as a view, and use the view name in place of the select statement.

We built one of the sets of sample databases that you'll find on the CD using the MySQL database system, which does not support embedded SELECT statements. When you look at those MySQL databases on the CD, you'll find that queries requiring an embedded SELECT statement are solved by saving a view and using the view name in the final solution query.

Watch out! First, when you decide to substitute a SELECT statement for a table name, be sure to include not only the columns you want to appear in the final result but also any linking columns needed to perform the JOIN. That's why you see both RecipeClassID and RecipeClassDescription in the embedded

statement. Just for fun, we gave RecipeClassDescription an alias name of ClassName in the embedded statement. As a result, the SELECT clause asks for ClassName rather than RecipeClassDescription. Note that the ON clause now references the correlation name of the embedded SELECT statement—RCFiltered—rather than the original name of the table or the correlation name we assigned the table inside the embedded SELECT statement.

If your database system has a very smart optimizer, defining your request this way should be just as fast as the previous example where the filter on RecipeClassDescription was applied via a WHERE clause *after* the JOIN. You would like to think that your database system, in order to answer your request most efficiently, would first filter the rows from Recipe_Classes before attempting to find any matching rows in Recipes. It could be much slower to first join all rows from Recipe_Classes with matching rows from Recipes and *then* apply the filter. If you find it's taking longer to solve this request than it should, moving the WHERE clause into a SELECT statement within the JOIN might force your database system to do the filtering on Recipe_Classes first.

Embedding JOINs within JOINs

Although you can solve many problems by linking only two tables, you'll often need to link three, four, or more tables to get all the data you require. For example, you might want to fetch all the relevant information about recipes—the type of recipe, the recipe name, and all the ingredients for the recipe—in one query. Figure 8–7 shows the tables required to answer this request.

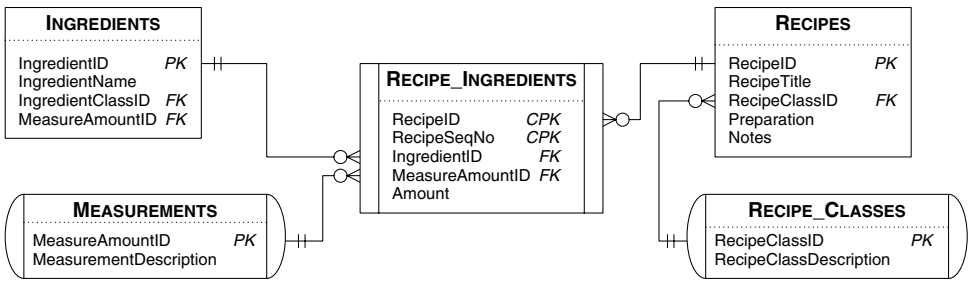


Figure 8–7 *The tables needed from the Recipes sample database to fetch all the information about recipes*

Looks like you need to get data from *five* different tables! Never fear—you can do this by constructing a more complex FROM clause, embedding JOIN clauses within JOIN clauses. Here's the trick: Everywhere you can specify a

table name, you can also specify an entire JOIN clause surrounded with parentheses. Figure 8–8 is a simplified version of Figure 8–4. (We've left off correlation name clauses and chosen the ON clause to form a simple JOIN of two tables.)

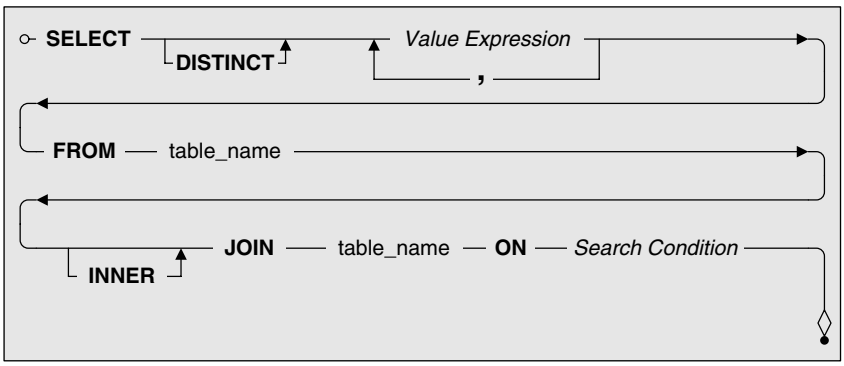


Figure 8–8 *A simple INNER JOIN of two tables*

To add a third table to the mix, just place an open parenthesis before the first table name, add a close parenthesis after the search condition, and insert INNER JOIN, a table name, the ON keyword, and another search condition. Figure 8–9 shows how to do this.

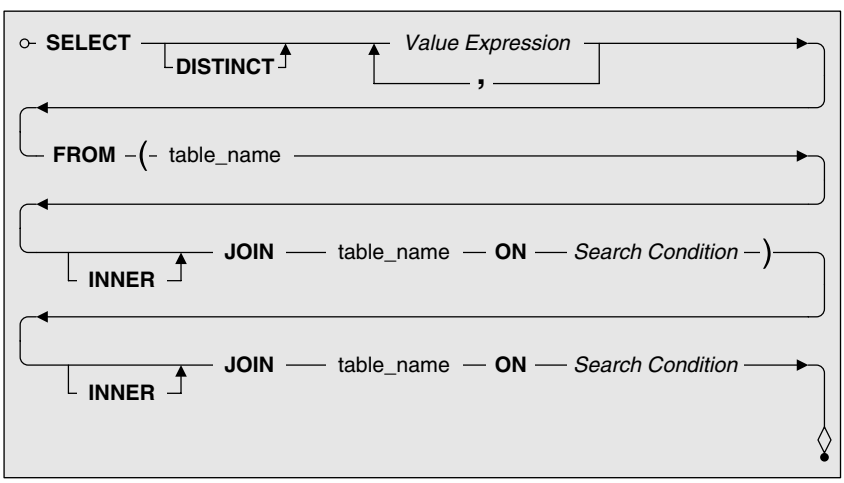


Figure 8–9 *A simple INNER JOIN of three tables*

If you think about it, the INNER JOIN of two tables inside the parentheses forms a logical table, or inner result set. This result set now takes the place of the first simple table name in Figure 8–8. You can continue this process of enclosing an entire JOIN clause in parentheses and then adding another JOIN keyword, table name, ON keyword, and search condition until you have all the result sets you need. Let's make a request that needs data from all the tables shown in Figure 8–7 and see how it turns out.

"I need the recipe type, recipe name, preparation instructions, ingredient names, ingredient step numbers, ingredient quantities, and ingredient measurements from my recipes database, sorted in step number sequence."

Translation

Select the recipe class description, recipe title, preparation instructions, ingredient name, recipe sequence number, amount, and measurement description from the recipe classes table joined with the recipes table on recipe class ID in the recipe classes table matching recipe class ID in the recipes table, then joined with the recipe ingredients table on recipe ID in the recipes table matching recipe ID in the recipe ingredients table, then joined with the ingredients table on ingredient ID in the ingredients table matching ingredient ID in the recipe ingredients table, and then finally joined with the measurements table on measurement amount ID in the measurements table matching measurement amount ID in the recipe ingredients table, order by recipe title and recipe sequence number

Clean up

Select ~~the~~ recipe class description, recipe title, preparation ~~instructions~~, ingredient name, recipe sequence number, amount, ~~and~~ measurement description
from ~~the~~ recipe classes ~~table~~
inner join~~ed with the~~ recipes ~~table~~
on recipe_classes.recipe class ID ~~in the recipe classes table matching~~
= recipes.recipe class ID ~~in the recipes table, then~~
inner join~~ed with the~~ recipe ingredients ~~table~~
on recipes.recipe ID ~~in the recipes table matching~~
= recipe_ ingredients.recipe ID ~~in the recipe ingredients table, then~~
inner join~~ed with the~~ ingredients ~~table~~
on ingredients.ingredient ID ~~in the ingredients table matching~~
= ingredients.ingredient ID ~~in the recipe ingredients table, and then finally~~ inner join~~ed with the~~ measurements ~~table~~

on measurements.measurement amount ID ~~in the measurements table matching~~
= recipe ingredients.measurement amount ID ~~in the recipe ingredients table,~~
order by recipe title ~~and~~ recipe sequence number

SQL

```
SELECT Recipe_Classes.RecipeClassDescription,
   Recipes.RecipeTitle, Recipes.Preparation,
   Ingredients.IngredientName,
   Recipe_Ingredients.RecipeSeqNo,
   Recipe_Ingredients.Amount,
   Measurements.MeasurementDescription
FROM (((Recipe_Classes
INNER JOIN Recipes
ON Recipe_Classes.RecipeClassID =
   Recipes.RecipeClassID)
INNER JOIN Recipe_Ingredients
ON Recipes.RecipeID =
   Recipe_Ingredients.RecipeID)
INNER JOIN Ingredients
ON Ingredients.IngredientID =
   Recipe_Ingredients.IngredientID)
INNER JOIN Measurements
ON Measurements.MeasureAmountID =
   Recipe_Ingredients.MeasureAmountID
ORDER BY RecipeTitle, RecipeSeqNo
```

Wow! Anyone care to jump in and add a filter for recipe class Main courses? If you said you need to add the WHERE clause just before the ORDER BY clause, you guessed the correct way to do it.

In truth, you can substitute an entire JOIN of two tables anywhere you could otherwise place only a table name. In Figure 8–9, we implied that you must first join the first table with the second table and then join that result with the third table. You could also join the second and third tables first (as long as the third table is, in fact, related to the second table and not the first one) and then perform the final join with the first table. Figure 8–10 (on page 260) shows this alternate method.

Let's look at the problem from a painting perspective. If you're trying to get pastel green, the mixing sequence doesn't matter that much. You can mix white with blue to get pastel blue and then mix in some yellow, or you can mix blue with yellow to get green and then add some white to get the final color.

To solve the request we just showed you using five tables, we could also have stated the SQL as follows.

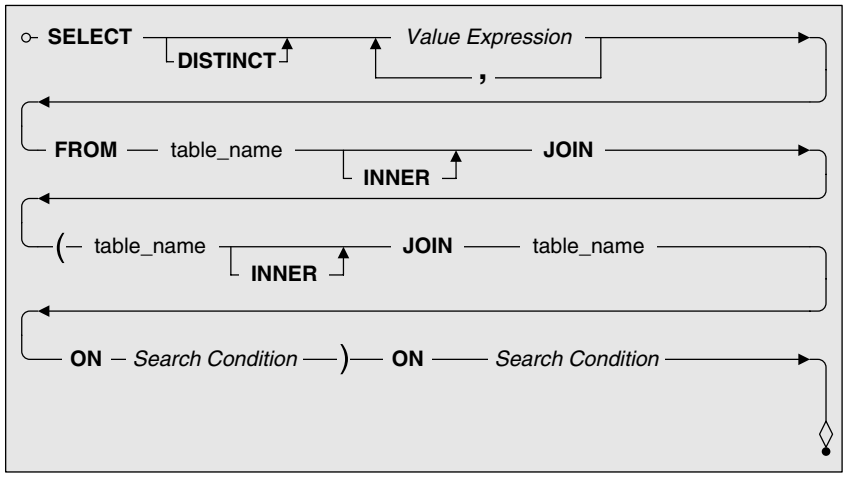


Figure 8–10 *Joining more than two tables in an alternate sequence*

```
SQL         SELECT Recipe_Classes.RecipeClassDescription,
               Recipes.RecipeTitle, Recipes.Preparation,
               Ingredients.IngredientName,
               Recipe_Ingredients.RecipeSeqNo,
               Recipe_Ingredients.Amount,
               Measurements.MeasurementDescription
            FROM Recipe_Classes
            INNER JOIN (((Recipes
            INNER JOIN Recipe_Ingredients
            ON Recipes.RecipeID =
               Recipe_Ingredients.RecipeID)
            INNER JOIN Ingredients
            ON Ingredients.IngredientID =
               Recipe_Ingredients.IngredientID)
            INNER JOIN Measurements
            ON Measurements.MeasureAmountID =
               Recipe_Ingredients.MeasureAmountID)
            ON Recipe_Classes.RecipeClassID =
               Recipes.RecipeClassID
            ORDER BY RecipeTitle, RecipeSeqNo
```

You need to be aware of this feature because you might run into this sort of construction either in queries others have written or in the SQL built for you by Query By Example software. Also, the optimizers in some database systems are sensitive to the sequence of the JOIN definitions. If you find your query using many JOINs is taking a long time to execute on a large database, you might be able to get it to run faster by changing the sequence of JOINs in your SQL statement. For simplicity, we'll build most of the examples later in this

chapter using a direct construction of JOINs by following a simple path from left to right and top to bottom, using the diagrams that you can find in Appendix B, Schema for the Sample Databases.

Check Those Relationships!

It should be obvious at this point that knowing the relationships between your tables is of utmost importance. When you find that the columns of data you need reside in different tables, you might need to construct a FROM clause as complicated as the one we just showed you to be able to gather all the pieces in a way that logically makes sense. If you don't know the relationships between your tables and the linking columns that form the relationships, you'll paint yourself into a corner!

In many cases, you might have to follow a path through several relationships to get the data you want. For example, let's simplify the previous request and just ask for recipe name and ingredient names.

"Show me the names of all my recipes and the names of all the ingredients for each of those recipes."

Translation
Select the recipe title and the ingredient name from the
recipes table joined with the recipe ingredients table
on recipe ID in the recipes table matching
recipe ID in the recipe ingredients table,
and then joined with the ingredients table
on ingredient ID in the ingredients table matching
ingredient ID in the recipe ingredients table

Clean Up
Select ~~the~~ recipe title ~~and the~~ ingredient name
from ~~the~~ recipes ~~table~~
inner join~~ed with the~~ recipe ingredients ~~table~~
on recipes.recipe ID ~~in the recipes table matching~~
= recipe_ingredients.recipe ID ~~in the recipe ingredients table, and then~~ inner join~~ed with the~~ ingredients ~~table~~
on ingredients.ingredient ID ~~in the ingredients table matching~~
= recipe_ingredients.ingredient ID ~~in the recipe ingredients table~~

SQL

```
SELECT Recipes.RecipeTitle,
   Ingredients.IngredientName
FROM (Recipes
INNER JOIN Recipe_Ingredients
```

```
ON Recipes.RecipeID =
   Recipe_Ingredients.RecipeID)
INNER JOIN Ingredients
ON Ingredients.IngredientID =
   Recipe_Ingredients.IngredientID
```

Did you notice that even though you don't need *any* columns from the Recipe_Ingredients table, you still must include it in the query? You must do so because the only way that Recipes and Ingredients are related is *through* the Recipe_Ingredients table.

Uses for INNER JOINs

Now that you have a basic understanding of the mechanics for constructing an INNER JOIN, let's look at some of the types of problems you can solve with it.

Find Related Rows

As you know, the most common use for an INNER JOIN is to link tables so that you can fetch columns from different tables that are related. Following is a sample list of the kinds of requests you can solve from the sample databases using an INNER JOIN.

"Show me the vendors and the products they supply to us."

"List employees and the customers for whom they booked an order."

"Display agents and the engagement dates they booked."

"List customers and the entertainers they booked."

"Find the entertainers who played engagements for customers Berg or Hallmark."

"Display buildings and all the classrooms in each building."

"List the faculty staff and the subject each teaches."

"Display bowling teams and the name of each team captain."

"List the bowling teams and all the team members."

"Show me the recipes that have beef or garlic."

"Display all the ingredients for recipes that contain carrots."

We'll show how to construct queries to answer requests like these (and more) in the Sample Statements section of this chapter.

Find Matching Values

A more esoteric use of an INNER JOIN is finding rows in two or more tables or result sets that match on one or more values that are *not* the related key values. Remember that in Chapter 7, Thinking in Sets, we promised to show you how to perform the equivalent of an INTERSECT using an INNER JOIN. Following is a small sample of just some of the requests you can solve using this technique.

"Show me customers and employees who have the same name."

"Show me customers and employees who live in the same city."

"Find all the customers who ordered a bicycle and also ordered a helmet."

"Find the agents and entertainers who live in the same postal code."

"List the entertainers who played engagements for customers Bonnicksen and Rosales."

"Show me the students and their teachers who have the same first name."

"Show me the students who have an average score of 85 or better in Art and who also have an average score of 85 or better in Computer Science."

"Find the bowlers who live in the same ZIP Code."

"Find the bowlers who had a raw score of 155 or better at both Thunderbird Lanes and Bolero Lanes."

"Find the ingredients that use the same default measurement amount."

"Show me the recipes that have beef and garlic."

The next section shows how to solve several problems like these.

Sample Statements

You now know the mechanics of constructing queries using INNER JOIN and have seen some of the types of requests you can answer with an INNER JOIN. Let's take a look at a fairly robust set of samples, all of which use INNER JOIN. These examples come from each of the sample databases, and they illustrate how you can use an INNER JOIN to fetch data from two tables, fetch data from more than two tables, and solve a problem using matching values.

We've also included sample result sets that would be returned by these operations and placed them immediately after the SQL syntax line. The name that appears immediately above a result set is the name we gave each query in the sample data on the companion CD you'll find bound into the back of the

book. We stored each query in the appropriate sample database (as indicated within the example) and prefixed the names of the queries relevant to this chapter with "CH08." You can follow the instructions in the Introduction of this book to load the samples onto your computer and try them.

> ❖ **Note** Because many of these examples use complex JOINs, your database system might choose a different way to solve these queries. For this reason, the first few rows we show you might not exactly match the result you obtain, but the total number of rows should be the same. To simplify the process, we have combined the Translation and Clean Up steps for all the following examples.

Two Tables

We'll start out with simple primary colors and show you sample requests that require an INNER JOIN on only two tables.

Sales Orders Database

"Display all products and their categories."

Translation/ Clean Up

Select category description ~~and~~ product name
from ~~the~~ categories ~~table~~
inner join~~ed with the~~ products ~~table~~
on categories.category ID ~~in the categories table matching~~
= products.category ID ~~in the products table~~

SQL

```
SELECT Categories.CategoryDescription,
   Products.ProductName
FROM Categories
INNER JOIN Products
   ON Categories.CategoryID = Products.CategoryID
```

CH08_Products_And_Categories (40 rows)

CategoryDescription	ProductName
Accessories	Dog Ear Cyclecomputer
Accessories	Dog Ear Helmet Mount Mirrors
Accessories	Viscount C-500 Wireless Bike Computer
Accessories	Kryptonite Advanced 2000 U-Lock
Accessories	Nikoma Lok-Tight U-Lock
Accessories	Viscount Microshell Helmet
Accessories	Viscount CardioSport Sport Watch
Accessories	Viscount Tru-Beat Heart Transmitter
Accessories	Dog Ear Monster Grip Gloves
<< more rows here >>	

Entertainment Agency Database

"Show me entertainers, the start and end dates of their contracts, and the contract price."

Translation/ Clean Up

Select entertainer stage name, start date, end date,
~~and~~ contract price from ~~the~~ entertainers ~~table~~
inner join~~ed with the~~ engagements ~~table~~
on entertainers.entertainer ID ~~in the entertainers table matching~~
= engagements.entertainer ID ~~in the engagements table~~

SQL

```
SELECT Entertainers.EntStageName,
   Engagements.StartDate, Engagements.EndDate,
   Engagements.ContractPrice
FROM Entertainers
INNER JOIN Engagements
   ON Entertainers.EntertainerID =
      Engagements.EntertainerID
```

CH08Entertainers_And_Contracts (111 rows)

EntStageName	StartDate	EndDate	ContractPrice
Carol Peacock Trio	2007-10-21	2007-10-21	$140.00
Carol Peacock Trio	2007-11-13	2007-11-19	$680.00
Carol Peacock Trio	2007-12-23	2007-12-26	$410.00
Carol Peacock Trio	2007-12-29	2008-01-07	$1,400.00
Carol Peacock Trio	2008-01-08	2008-01-08	$320.00
Carol Peacock Trio	2008-01-22	2008-01-30	$1,670.00
Carol Peacock Trio	2008-02-11	2008-02-19	$1,670.00
Carol Peacock Trio	2008-02-25	2008-02-28	$770.00
Topazz	2007-09-11	2007-09-18	$770.00
<< more rows here >>			

School Scheduling Database

"List the subjects taught on Wednesday."

Translation/ Clean Up

Select subject name
from ~~the~~ subjects ~~table~~
inner join~~ed with the~~ classes ~~table~~
on subjects.subject ID ~~in the subjects table matching~~
= classes.subject ID ~~in the classes table~~
where Wednesday schedule ~~is~~ = true

SQL

```
SELECT DISTINCT Subjects.SubjectName
FROM Subjects
INNER JOIN Classes
  ON Subjects.SubjectID
  = Classes.SubjectID
WHERE Classes.WednesdaySchedule = -1
```

❖ **Note** Because several sections of the same class might be scheduled on the same day of the week, we included the DISTINCT keyword to eliminate the duplicates. Some databases do support a TRUE keyword, but we chose to use a more universal "integer with all bits on" value: –1. If your database system stores a true/false value as a single bit, you can also test for a true value of 1. A false value is always the number zero (0).

CH08_Subjects_On_Wednesday (45 rows)

SubjectName
Advanced English Grammar
Art History
Biological Principles
Business Tax Accounting
Chemistry
Composition—Fundamentals
Composition—Intermediate
Computer Art
Database Management
<< more rows here >>

Bowling League Database

"Display bowling teams and the name of each team captain."

Translation/ Clean Up

Select team name ~~and~~ captain full name
from ~~the~~ teams ~~table~~
inner join~~ed with the~~ bowlers ~~table~~
on team captain ID ~~equals~~ = bowler ID

SQL

```
SELECT Teams.TeamName, (Bowlers.BowlerLastName
   || ', ' || Bowlers.BowlerFirstName) AS CaptainName
FROM Teams
INNER JOIN Bowlers
  ON Teams.CaptainID = Bowlers.BowlerID
```

CH08_Teams_And_Captains (10 rows)

TeamName	CaptainName
Marlins	Fournier, David
Sharks	Patterson, Ann
Terrapins	Viescas, Carol
Barracudas	Sheskey, Richard
Dolphins	Viescas, Suzanne
Orcas	Thompson, Sarah
Manatees	Viescas, Michael
Swordfish	Rosales, Joe
Huckleberrys	Viescas, David
MintJuleps	Hallmark, Alaina

Recipes Database

"Show me the recipes that have beef or garlic."

Translation/ Clean Up

Select ~~unique~~ distinct recipe title
from ~~the~~ recipes ~~table~~
join~~ed with the~~ recipe ingredients ~~table~~
on recipes.recipe ID ~~in the recipes table matching~~
= recipe_ingredients.recipe ID ~~in the recipe ingredients table~~
where ingredient ID ~~is~~ in ~~the list of beef and garlic IDs~~ (1, 9)

SQL

```
SELECT DISTINCT Recipes.RecipeTitle
FROM Recipes
INNER JOIN Recipe_Ingredients
  ON Recipes.RecipeID = Recipe_Ingredients.RecipeID
WHERE Recipe_Ingredients.IngredientID IN (1, 9)
```

❖ **Note** Because some recipes might have both beef and garlic, we added the DISTINCT keyword to eliminate potential duplicate rows.

CH08_Beef_Or_Garlic_Recipes (5 rows)

RecipeTitle
Asparagus
Garlic Green Beans
Irish Stew
Pollo Picoso
Roast Beef

More Than Two Tables

Next, let's add some spice by making requests that require a JOIN of more than two tables.

Sales Orders Database

"Find all the customers who ever ordered a bicycle helmet."

Translation/ Clean Up

Select customer first name, customer last name
from ~~the~~ customers ~~table~~
inner join~~ed with the~~ orders ~~table~~
on customers.customer ID ~~in the customers table matching~~
= orders.customer ID ~~in the orders table,~~
~~then~~ inner join~~ed with the~~ order details ~~table~~
on orders.order number ~~in the orders table matching~~
= order_details.order number ~~in the order details table, then~~
inner join~~ed with the~~ products ~~table~~
on products.product number ~~in the products table matching~~
= order_ details.product number ~~in the order details table~~
where product name ~~contains~~ LIKE '%Helmet%'

SQL

```
SELECT DISTINCT Customers.CustFirstName,
   Customers.CustLastName
FROM ((Customers
INNER JOIN Orders
  ON Customers.CustomerID = Orders.CustomerID)
INNER JOIN Order_Details
  ON Orders.OrderNumber =
     Order_Details.OrderNumber)
INNER JOIN Products
  ON Products.ProductNumber =
     Order_Details.ProductNumber
WHERE Products.ProductName LIKE '%Helmet%'
```

❖ **Caution** If your database system is case sensitive when performing searches in character fields, you must be careful to enter the search criteria using the correct case for the letters. For example, in many database systems, 'helmet' is not the same as 'Helmet'.

❖ **Note** Because a customer might have ordered a helmet more than once, we included the DISTINCT keyword to eliminate duplicate rows.

CH08_Customers_Who_Ordered_Helmets (25 rows)

CustFirstName	CustLastName
Andrew	Cencini
Angel	Kennedy
Caleb	Viescas
Darren	Gehring
David	Smith
Dean	McCrae
Estella	Pundt
Gary	Hallmark
Jim	Wilson
John	Viescas
<< more rows here >>	

Entertainment Agency Database

"Find the entertainers who played engagements for customers Berg or Hallmark."

Translation/ Clean Up

Select ~~unique~~ distinct entertainer stage name
from ~~the~~ entertainers ~~table~~
inner join~~ed with the~~ engagements ~~table~~
on entertainers.entertainer ID
~~in the entertainers table matching~~
= engagements.entertainer ID ~~in the engagements table,~~
~~then~~ inner join~~ed with the~~ customers ~~table~~
on customers.customer ID ~~in the customers table matching~~
= engagements.customer ID ~~in the engagements table~~
where ~~the~~ customer last name ~~is~~ = 'Berg'
or ~~the~~ customer last name ~~is~~ = 'Hallmark'

SQL

```
SELECT DISTINCT Entertainers.EntStageName
FROM (Entertainers
INNER JOIN Engagements
   ON Entertainers.EntertainerID =
      Engagements.EntertainerID)
INNER JOIN Customers
   ON Customers.CustomerID =
      Engagements.CustomerID
WHERE Customers.CustLastName = 'Berg'
   OR Customers.CustLastName = 'Hallmark'
```

CH08_Entertainers_For_Berg_OR_Hallmark (8 rows)

EntStageName
Carol Peacock Trio
Coldwater Cattle Company
Country Feeling
Jim Glynn
JV & the Deep Six
Modern Dance
Susan McLain
Topazz

Bowling League Database

"List all the tournaments, the tournament matches, and the game results."

Translation/ Clean Up

Select tourney ID, tourney location, match ID, lanes, odd lane team, even lane team, game number, game winner
from ~~the~~ tournaments ~~table~~
inner join~~ed with the~~ tourney matches ~~table~~
on tournaments.tourney ID ~~in the tournaments table matching~~
= tourney_matches.tourney ID ~~in the tourney matches table,~~
~~then~~ inner join~~ed with the~~ teams ~~table aliased~~ as odd team
on oddteam.team ID ~~in the odd team table matches~~
= teams.odd lane team ID ~~in the teams table,~~
~~then~~ inner join~~ed with the~~ teams ~~table aliased~~ as even team
on eventeam.team ID ~~in the even team table matches~~
= tourney_matches.even lane team ID
~~in the tourney matches table,~~
~~then~~ inner join~~ed with the~~ match games ~~table~~
on match_games.match ID ~~in the match games table matches~~
= tourney_matches.match ID ~~in the tourney matches table,~~
~~then~~ inner join~~ed with the~~ teams ~~table aliased~~ as winner
on winner.team ID ~~in the winner table matches~~
= match_games.winning team ID ~~in the match games table~~

```
SQL         SELECT Tournaments.TourneyID AS Tourney,
            Tournaments.TourneyLocation AS Location,
            Tourney_Matches.MatchID, Tourney_Matches.Lanes,
            OddTeam.TeamName AS OddLaneTeam,
            EvenTeam.TeamName AS EvenLaneTeam,
            Match_Games.GameNumber AS GameNo, Winner.TeamName AS
            Winner
            FROM Tournaments
              INNER JOIN Tourney_Matches
                ON Tournaments.TourneyID
                = Tourney_Matches.TourneyID
              INNER JOIN Teams AS OddTeam
                ON OddTeam.TeamID
                = Tourney_Matches.OddLaneTeamID
              INNER JOIN Teams AS EvenTeam
                ON EvenTeam.TeamID
                = Tourney_Matches.EvenLaneTeamID
              INNER JOIN Match_Games
                ON Match_Games.MatchID
                = Tourney_Matches.MatchID
              INNER JOIN Teams AS Winner
                ON Winner.TeamID
                = Match_Games.WinningTeamID
```

❖ **Note** This is a really fun query because it requires *three* copies of one table (Teams) to get the job done. We had to assign correlation names to at least two of the tables to keep everything legal, but we went ahead and gave them all alias names to reflect their specific roles in the query. Also, when we constructed the SQL, we didn't exactly follow the structure of the Translation/Clean Up step. We did this to demonstrate that you can define the nested JOINs in any way you like as long as you keep the relationships straight.

CH08_Tournament_Match_Game_Results (168 rows)

Tourney	Location	MatchID	Lanes	OddLane Team	EvenLane Team	GameNo	Winner
1	Red Rooster Lanes	1	01-02	Marlins	Sharks	1	Marlins
1	Red Rooster Lanes	1	01-02	Marlins	Sharks	2	Sharks
1	Red Rooster Lanes	1	01-02	Marlins	Sharks	3	Marlins
1	Red Rooster Lanes	2	03-04	Terrapins	Barracudas	1	Terrapins
1	Red Rooster Lanes	2	03-04	Terrapins	Barracudas	2	Barracudas
1	Red Rooster Lanes	2	03-04	Terrapins	Barracudas	3	Terrapins
1	Red Rooster Lanes	3	05-06	Dolphins	Orcas	1	Dolphins
1	Red Rooster Lanes	3	05-06	Dolphins	Orcas	2	Orcas
1	Red Rooster Lanes	3	05-06	Dolphins	Orcas	3	Dolphins
<< more rows here >>							

❖ **Note** Although the records appear to be sorted by tournament and match, this is simply the sequence in which the database system we used (in this case, Microsoft Office Access) chose to return the records. If you want to ensure that the records are sorted in a specific sequence, you must supply an ORDER BY clause.

Recipes Database

"Show me the main course recipes and list all the ingredients."

Translation/ Clean Up

Select recipe title, ingredient name,
measurement description, ~~and~~ amount
from ~~the~~ recipe classes ~~table~~
inner join~~ed with the~~ recipes ~~table~~
on recipes.recipe class ID ~~in the recipes table matches~~
= recipe_classes.recipe class ID ~~in the recipe classes table,~~
~~then~~ inner join~~ed with the~~ recipe ingredients ~~table~~
on recipes.recipe ID ~~in the recipes table matches~~
= recipe_ingredients.recipe ID ~~in the recipe ingredients table,~~
~~then~~ inner join~~ed with the~~ ingredients ~~table~~
on ingredients.ingredient ID ~~in the ingredients table matches~~
= recipe_ingredients.ingredient ID ~~in the recipe ingredients table,~~
~~and finally~~ inner join~~ed with the~~ measurements ~~table~~
on measurements.measure amount ID
~~in the measurements table matches~~
= recipe_ingredients.measure amount ID
~~in the recipe ingredients table,~~
where recipe class description ~~is~~ = 'Main course'

SQL

```
SELECT Recipes.RecipeTitle,
   Ingredients.IngredientName,
   Measurements.MeasurementDescription,
   Recipe_Ingredients.Amount
FROM (((Recipe_Classes
INNER JOIN Recipes
ON Recipes.RecipeClassID =
   Recipe_Classes.RecipeClassID)
INNER JOIN Recipe_Ingredients
ON Recipes.RecipeID =
   Recipe_Ingredients.RecipeID)
INNER JOIN Ingredients
ON Ingredients.IngredientID =
   Recipe_Ingredients.IngredientID)
INNER JOIN Measurements
ON Measurements.MeasureAmountID =
   Recipe_Ingredients.MeasureAmountID
WHERE Recipe_Classes.RecipeClassDescription =
   'Main course'
```

❖ **Caution** You can find a MeasureAmountID in both the Ingredients and the Recipe_Ingredients tables. If you define the final JOIN on MeasureAmountID using the Ingredients table instead of the Recipe_Ingredients table, you'll get the default measurement for the ingredient rather than the one specified for the ingredient in the recipe.

CH08_Main_Course_Ingredients (53 rows)

RecipeTitle	IngredientName	MeasurementDescription	Amount
Irish Stew	Beef	Pound	1
Irish Stew	Onion	Whole	2
Irish Stew	Potato	Whole	4
Irish Stew	Carrot	Whole	6
Irish Stew	Water	Quarts	4
Irish Stew	Guinness Beer	Ounce	12
Fettuccini Alfredo	Fettuccini Pasta	Ounce	16
Fettuccini Alfredo	Vegetable Oil	Tablespoon	1
Fettuccini Alfredo	Salt	Teaspoon	3
<< more rows here >>			

Looking for Matching Values

Finally, let's add a third dimension to the picture. This last set of examples shows requests that use a JOIN on common values from two or more result sets or tables. (If your database supports the INTERSECT keyword, you can also solve many of these problems by intersecting the result sets.)

Sales Orders Database

"Find all the customers who ordered a bicycle and also ordered a helmet."

This request seems simple enough—perhaps too simple. Let's ask it a different way so that it's clearer what we need the database to do.

"Find all the customers who ordered a bicycle, then find all the customers who ordered a helmet, and finally list the common customers so that we know who ordered both a bicycle and a helmet."

Translation 1

Select customer first name and customer last name from those common to the set of customers who ordered bicycles and the set of customers who ordered helmets

Translation 2/ Clean Up

Select customer first name ~~and~~ customer last name from
(Select ~~unique~~ distinct customer name,
customer first name, customer last name
from ~~the~~ customers ~~table~~
inner join~~ed with the~~ orders ~~table~~
on customers.customer ID ~~in the customers table matches~~
= orders.customer ID ~~in the orders table,~~
~~then~~ inner join~~ed with the~~ order details ~~table~~
on orders.order number ~~in the orders table matches~~
= order_ details.order number ~~in the order details table,~~
~~then~~ inner join~~ed with the~~ products ~~table~~
on products.product number ~~in the products table matches~~
= order_details.product number ~~in the order details table~~
where product name ~~contains~~ LIKE '%Bike') as cust bikes
inner join~~ed with~~
(Select ~~unique~~ distinct customer ID
from ~~the~~ customers ~~table~~
inner join~~ed with the~~ orders ~~table~~
on customers.customer ID ~~in the customers table matches~~
= orders.customer ID ~~in the orders table,~~
~~then~~ inner join~~ed with the~~ order details ~~table~~
on orders.order number ~~in the orders table matches~~
= order_ details.order number ~~in the order details table,~~
~~then~~ join~~ed with the~~ products ~~table~~
on products.product number ~~in the products table matches~~
= order_details.product number ~~in the order details table~~
where product name ~~contains~~ LIKE '%Helmet') as cust helmets
on cust bikes.customer ID ~~in the cust bikes table matches~~
= cust helmets.customer ID ~~in the cust helmets table~~

SQL

```
SELECT CustBikes.CustFirstName,
   CustBikes.CustLastName
FROM
   (SELECT DISTINCT Customers.CustomerID,
      Customers.CustFirstName,
      Customers.CustLastName
    FROM ((Customers
    INNER JOIN Orders
    ON Customers.CustomerID
      = Orders.CustomerID)
    INNER JOIN Order_Details
    ON Orders.OrderNumber =
      Order_Details.OrderNumber)
    INNER JOIN Products
    ON Products.ProductNumber =
      Order_Details.ProductNumber
    WHERE Products.ProductName LIKE '%Bike')
  AS CustBikes
INNER JOIN
   (SELECT DISTINCT Customers.CustomerID
    FROM ((Customers
    INNER JOIN Orders
    ON Customers.CustomerID = Orders.CustomerID)
    INNER JOIN Order_Details
    ON Orders.OrderNumber =
       Order_Details.OrderNumber)
    INNER JOIN Products
    ON Products.ProductNumber =
       Order_Details.ProductNumber
    WHERE Products.ProductName LIKE '%Helmet')
  AS CustHelmets
ON CustBikes.CustomerID =
   CustHelmets.CustomerID
```

❖ **Note** We simplified the second embedded SELECT statement to fetch only the CustomerID because that's the only column we need for the INNER JOIN of the two sets to work. We could have actually eliminated the JOIN to the Customers table and fetched the CustomerID from the Orders table. Remember that you can think of a SELECT Statement embedded in a FROM clause as a "logical table," and we assigned a unique name to each statement so that we could write the final ON clause.

You could also solve this problem as the INTERSECT of the two sets, but you would need to include all the output columns in both of the result sets that you intersect. Quite frankly, this might not be the best way to solve this problem. We'll show you how to solve this problem more efficiently in Chapter 11, when we teach you how to use subqueries.

CH08_Customers_Both_Bikes_And_Helmets (21 rows)

CustFirstName	CustLastName
William	Thompson
Robert	Brown
Dean	McCrae
John	Viescas
Mariya	Sergienko
Neil	Patterson
Andrew	Cencini
Angel	Kennedy
Liz	Keyser
Rachel	Patterson
<< more rows here >>	

Entertainment Agency Database

"List the entertainers who played engagements for both customers Berg and Hallmark."

As you saw earlier, solving for Berg *or* Hallmark is easy. Let's phrase the request a different way so that it's clearer what we need the database to do for us.

"Find all the entertainers who played an engagement for Berg, then find all the entertainers who played an engagement for Hallmark, and finally list the common entertainers so that we know who played an engagement for both."

Translation 1

Select entertainer stage name from those common to the set of entertainers who played for Berg and the set of entertainers who played for Hallmark

Translation 2/ Clean Up

Select entertainer stage name
from (Select ~~unique~~ distinct entertainer stage name
from ~~the~~ entertainers ~~table~~
inner join~~ed with the~~ engagements ~~table~~
on entertainers.entertainer ID ~~in the entertainers table matches~~
= engagements.entertainer ID ~~in the engagements table, then~~ inner join~~ed with the~~ customers ~~table~~
on customers.customer ID ~~in the customers table matches~~
= engagements.customer ID ~~in the engagements table~~
where customer last name ~~is~~ = 'Berg') as entberg
inner join~~ed with~~
(Select ~~unique~~ distinct entertainer stage names
from ~~the~~ entertainers ~~table~~
inner join~~ed with the~~ engagements ~~table~~
on entertainers.entertainer ID ~~in the entertainers table matches~~
= engagements.entertainer ID ~~in the engagements table, then~~
join~~ed with the~~ customers ~~table~~
oncustomers.customer ID ~~in the customers table matches~~
= engagements.customer ID ~~in the engagements table~~
where customer last name ~~is~~ = 'Hallmark') as enthallmark
on entberg.entertainer ID ~~in the entberg table matches~~
= enthallmark.entertainer ID ~~in the enthallmark table~~

SQL

```
SELECT EntBerg.EntStageName
FROM
   (SELECT DISTINCT Entertainers.EntertainerID,
      Entertainers.EntStageName
    FROM (Entertainers
    INNER JOIN Engagements
    ON Entertainers.EntertainerID =
       Engagements.EntertainerID)
    INNER JOIN Customers
    ON Customers.CustomerID =
       Engagements.CustomerID
    WHERE Customers.CustLastName = 'Berg')
  AS EntBerg
INNER JOIN
   (SELECT DISTINCT Entertainers.EntertainerID,
      Entertainers.EntStageName
    FROM (Entertainers
    INNER JOIN Engagements
    ON Entertainers.EntertainerID =
       Engagements.EntertainerID)
    INNER JOIN Customers
    ON Customers.CustomerID =
       Engagements.CustomerID
    WHERE Customers.CustLastName = 'Hallmark')
  AS EntHallmark
ON EntBerg.EntertainerID =
   EntHallmark.EntertainerID
```

CH08_Entertainers_Berg_AND_Hallmark(4 rows)

EntStageName
Carol Peacock Trio
JV & the Deep Six
Modern Dance
Country Feeling

❖ **Note** This is another example of a request that can also be solved with INTERSECT. It can also be solved more efficiently with subqueries, which you'll learn about in Chapter 11.

School Scheduling Database

"Show me the students and teachers who have the same first name."

Translation/ Clean Up

Select student full name ~~and~~ staff full name
from ~~the~~ students ~~table~~
inner join~~ed with the~~ staff ~~table~~
on students.first name ~~in the students table matches~~
= staff.first name ~~in the staff table~~

SQL

```
SELECT (Students.StudFirstName || ' ' ||
   Students.StudLastName) AS StudFullName,
   (Staff.StfFirstName || ' ' ||
   Staff.StfLastName) AS StfFullName
FROM Students
INNER JOIN Staff
ON Students.StudFirstName = Staff.StfFirstName
```

CH08_Students_Staff_Same_FirstName (2 rows)

StudFullName	StfFullName
Michael Viescas	Michael Hernandez
David Hamilton	David Smith

Bowling League Database

"Find the bowlers who had a raw score of 170 or better at both Thunderbird Lanes and Bolero Lanes."

Yes, this is another "solve an intersection with a JOIN" problem. Let's ask it a different way so that it's clearer what we need the database to do for us.

"Find all the bowlers who had a raw score of 170 or better at Thunderbird Lanes, then find all the bowlers who had a raw score of 170 or better at Bolero Lanes, and finally list the common bowlers so that we know who had good scores at both bowling alleys."

Translation 1

Select bowler full name from those common to the set of bowlers who have a score of 170 or better at Thunderbird Lanes and the set of bowlers who have a score of 170 or better at Bolero Lanes

Translation 2/ Clean Up

Select bowler full name
from (Select ~~unique~~ distinct bowler ID ~~and~~ bowler full name
from ~~the~~ bowlers ~~table~~
inner join~~ed with the~~ bowler scores ~~table~~
on bowlers.bowler ID ~~in the bowlers table matches~~
= bowler_scores.bowler ID ~~in the bowler scores table,~~
~~then~~ inner join~~ed with the~~ tourney matches ~~table~~
on tourney_matches.match ID
~~in the tourney matches table matches~~
= bowler_scores.match ID ~~in the bowler scores table,~~
~~and finally~~ inner join~~ed with the~~ tournaments ~~table~~
on tournaments.tourney ID ~~in the tournaments table matches~~
= tourney_matches.tourney ID ~~in the tourney matches table~~
where tourney location ~~is~~ = 'Thunderbird Lanes'
and raw score ~~is greater than or equal to~~ >= 170) as bowlertbird
inner join~~ed with~~
(Select ~~unique~~ distinct bowler ID ~~and~~ bowler full name
from ~~the~~ bowlers ~~table~~
inner join~~ed with the~~ bowler scores ~~table~~
on bowlers.bowler ID ~~in the bowlers table matches~~
= bower_scores.bowler ID ~~in the bowler scores table,~~
~~then~~ inner join~~ed with the~~ tourney matches ~~table~~
on tourney_matches.match ID
~~in the tourney matches table matches~~
= bowler_scores.match ID ~~in the bowler scores table,~~
~~and finally~~ inner join~~ed with the~~ tournaments ~~table~~

on tournaments.tourney ID ~~in the tournaments table matches~~
= tourney_matches.tourney ID ~~in the tourney matches table~~
where tourney location ~~is~~ = 'Bolero Lanes'
and raw score ~~is greater than or equal to~~ >= 170)
  as bowlerbolero
on bowlertbird.bowler ID ~~in the bowlertbird table matches~~
= bowlerbolero.bowler ID ~~in the bowlerbolero table~~

SQL

```
SELECT BowlerTbird.BowlerFullName
FROM
   (SELECT DISTINCT Bowlers.BowlerID,
      (Bowlers.BowlerLastName || ', ' ||
      Bowlers.BowlerFirstName) AS BowlerFullName
    FROM ((Bowlers
    INNER JOIN Bowler_Scores
    ON Bowlers.BowlerID = Bowler_Scores.BoWlerID)
    INNER JOIN Tourney_Matches
    ON Tourney_Matches.MatchID =
      Bowler_Scores.MatchID)
    INNER JOIN Tournaments
    ON Tournaments.TourneyID =
      Tourney_Matches.TourneyID
    WHERE Tournaments.TourneyLocation =
      'Thunderbird Lanes'
    AND Bowler_Scores.RawScore >= 170)
  AS BowlerTbird
INNER JOIN
   (SELECT DISTINCT Bowlers.BowlerID,
      (Bowlers.BowlerLastName || ', ' ||
      Bowlers.BowlerFirstName) AS BowlerFullName
    FROM ((Bowlers
    INNER JOIN Bowler_Scores
    ON Bowlers.BowlerID = Bowler_Scores.BowlerID)
    INNER JOIN Tourney_Matches
    ON Tourney_Matches.MatchID =
      Bowler_Scores MatchID)
    INNER JOIN Tournaments
    ON Tournaments.TourneyID =
      Tourney_Matches.TourneyID
    WHERE Tournaments.TourneyLocation =
      'Bolero Lanes'
    AND Bowler_Scores.RawScore >= 170)
  AS BowlerBolero
ON BowlerTbird.BowlerID = BowlerBolero.BowlerID
```

❖ **Note** Because a bowler might have had a high score at either bowling alley more than once, we added the DISTINCT keyword to eliminate the duplicates. Again, this is a problem that might be better solved with subqueries, which you'll learn about in Chapter 11.

CH08_Good_Bowlers_TBird_And_Bolero (11 rows)

BowlerFullName
Kennedy, John
Patterson, Neil
Kennedy, Angel
Patterson, Kathryn
Viescas, John
Viescas, Caleb
Thompson, Sarah
Thompson, Mary
Thompson, William
Patterson, Rachel
Pundt, Steve

Recipes Database

"Display all the ingredients for recipes that contain carrots."

Translation/ Clean Up

Select recipe ID, recipe title, ~~and~~ ingredient name
from ~~the~~ recipes ~~table~~
inner join~~ed with the~~ recipe ingredients ~~table~~
on recipes.recipe ID ~~in the recipes table matches~~
= recipe_ingredients.recipe ID ~~in the recipe ingredients table,~~
inner join~~ed with the~~ ingredients ~~table~~
on ingredients.ingredient ID ~~in the ingredients table matches~~
= recipe_ingredients.ingredient ID ~~in the recipe ingredients table,~~
~~then finally~~ inner join~~ed with~~
(Select recipe ID from ~~the~~ ingredients ~~table~~
inner join~~ed with the~~ recipe ingredients ~~table~~
on ingredients.ingredient ID ~~in the ingredients table matches~~
= recipe_ingredients.ingredient ID ~~in the recipe ingredients table~~
where ingredient name ~~is~~ = 'Carrot') as carrots
on recipes.recipe ID ~~in the recipes table matches~~
= carrots.recipe ID ~~in the carrots table~~

SQL

```
SELECT Recipes.RecipeID, Recipes.RecipeTitle,
   Ingredients.IngredientName
FROM ((Recipes
INNER JOIN Recipe_Ingredients
ON Recipes.RecipeID =
   Recipe_Ingredients.RecipeID)
INNER JOIN Ingredients
ON Ingredients.IngredientID =
   Recipe_Ingredients.IngredientID)
INNER JOIN
   (SELECT Recipe_Ingredients.RecipeID
    FROM Ingredients
    INNER JOIN Recipe_Ingredients
    ON Ingredients.IngredientID =
       Recipe_Ingredients.IngredientID
    WHERE Ingredients.IngredientName = 'Carrot')
  AS Carrots
ON Recipes.RecipeID = Carrots.RecipeID
```

❖ **Note** This request can be solved more simply with a subquery. We'll show you how to do that in Chapter 11.

CH08_Recipes_Containing_Carrots (16 rows)

RecipeID	RecipeTitle	IngredientName
1	Irish Stew	Beef
1	Irish Stew	Onion
1	Irish Stew	Potato
1	Irish Stew	Carrot
1	Irish Stew	Water
1	Irish Stew	Guinness Beer
14	Salmon Filets in Parchment Paper	Salmon
14	Salmon Filets in Parchment Paper	Carrot
14	Salmon Filets in Parchment Paper	Leek
14	Salmon Filets in Parchment Paper	Red Bell Pepper
14	Salmon Filets in Parchment Paper	Butter
<< more rows here >>		

SUMMARY

In this chapter, we thoroughly discussed how to link two or more tables or result sets on matching values. We began by defining the concept of a JOIN, and then we went into the details about forming an INNER JOIN. We discussed what is "legal" to use as the criteria for a JOIN but cautioned you about making nonsensical JOINs.

We started out simply with examples joining two tables. We next showed how to assign correlation (alias) names to tables within your FROM clause. You might want to do this for convenience—or you might be required to assign correlation names when you include the same table more than once or use an embedded SELECT statement.

We showed how to replace a reference to a table with a SELECT statement within your FROM clause. We next showed how to extend your horizons by joining more than two tables or result sets. We wrapped up the discussion of the syntax of an INNER JOIN by reemphasizing the importance of having a good database design and understanding how your tables are related.

We discussed a number of reasons why INNER JOINs are useful and gave you specific examples. The rest of the chapter provided more than a dozen examples of using INNER JOIN. We broke these examples into JOINs on two tables, JOINs on more than two tables, and JOINs on matching values. In the next chapter, we'll explore another variant of JOIN—an OUTER JOIN.

The following section presents a number of requests to work out on your own.

Problems for You to Solve

Below, we show you the request statement and the name of the solution query in the sample databases. If you want some practice, you can work out the SQL you need for each request and then check your answer with the query we saved in the samples. Don't worry if your syntax doesn't exactly match the syntax of the queries we saved—as long as your result set is the same.

Sales Orders Database

1. *"List customers and the dates they placed an order, sorted in order date sequence."*
   (Hint: The solution requires a JOIN of two tables.)
   You can find the solution in CH08_Customers_And_OrderDates (944 rows).
2. *"List employees and the customers for whom they booked an order."*
   (Hint: The solution requires a JOIN of more than two tables.)
   You can find the solution in CH08_Employees_And_Customers (211 rows).
3. *"Display all orders, the products in each order, and the amount owed for each product, in order number sequence."*
   (Hint: The solution requires a JOIN of more than two tables.)
   You can find the solution in CH08_Orders_With_Products (3,975 rows).
4. *"Show me the vendors and the products they supply to us for products that cost less than $100."*
   (Hint: The solution requires a JOIN of more than two tables.)
   You can find the solution in CH08_Vendors_And_Products_Less_Than_100 (66 rows).
5. *"Show me customers and employees who have the same last name."*
   (Hint: The solution requires a JOIN on matching values.)
   You can find the solution in CH08_Customers_Employees_Same_LastName (16 rows).

6. *"Show me customers and employees who live in the same city."*
   (Hint: The solution requires a JOIN on matching values.)
   You can find the solution in CH08_Customers_Employees_Same_City (10 rows).

Entertainment Agency Database

1. *"Display agents and the engagement dates they booked, sorted by booking start date."*
   (Hint: The solution requires a JOIN of two tables.)
   You can find the solution in CH08_Agents_Booked_Dates (111 rows).
2. *"List customers and the entertainers they booked."*
   (Hint: The solution requires a JOIN of more than two tables.)
   You can find the solution in CH08_Customers_Booked_Entertainers (75 rows).
3. *"Find the agents and entertainers who live in the same postal code."*
   (Hint: The solution requires a JOIN on matching values.)
   You can find the solution in CH08_Agents_Entertainers_Same_Postal (10 rows).

School Scheduling Database

1. *"Display buildings and all the classrooms in each building."*
   (Hint: The solution requires a JOIN of two tables.)
   You can find the solution in CH08_Buildings_Classrooms (44 rows).
2. *"List students and all the classes in which they are currently enrolled."*
   (Hint: The solution requires a JOIN of more than two tables.)
   You can find the solution in CH08_Student_Enrollments (33 rows).
3. *"List the faculty staff and the subject each teaches."*
   (Hint: The solution requires a JOIN of more than two tables.)
   You can find the solution in CH08_Staff_Subjects (110 rows).
4. *"Show me the students who have a grade of 85 or better in art and who also have a grade of 85 or better in any computer course."*
   (Hint: The solution requires a JOIN on matching values.)
   You can find the solution in CH08_Good_Art_CS_Students (1 row).

Bowling League Database

1. *"List the bowling teams and all the team members."*
   (Hint: The solution requires a JOIN of two tables.)
   You can find the solution in CH08_Teams_And_Bowlers (32 rows).
2. *"Display the bowlers, the matches they played in, and the bowler game scores."*
   (Hint: The solution requires a JOIN of more than two tables.)
   You can find the solution in CH08_Bowler_Game_Scores (1,344 rows).

3. *"Find the bowlers who live in the same ZIP Code."*
   (Hint: The solution requires a JOIN on matching values.)
   You can find the solution in CH08_Bowlers_Same_ZipCode (92 rows).

Recipes Database

1. *"List all the recipes for salads."*
   (Hint: The solution requires a JOIN of two tables.)
   You can find the solution in CH08_Salads (1 row).
2. *"List all recipes that contain a dairy ingredient."*
   (Hint: The solution requires a JOIN of more than two tables.)
   You can find the solution in CH08_Recipes_Containing_Dairy (2 rows).
3. *"Find the ingredients that use the same default measurement amount."*
   (Hint: The solution requires a JOIN on matching values.)
   You can find the solution in CH08_Ingredients_Same_Measure (628 rows).
4. *"Show me the recipes that have beef and garlic."*
   (Hint: The solution requires a JOIN on matching values.)
   You can find the solution in CH08_Beef_And_Garlic_Recipes (1 row).

This page intentionally left blank

9

OUTER JOINs

"The only difference between a problem and a solution is people understand the solution."
—Charles Franklin Kettering
Inventor, 1876–1958

Topics Covered in This Chapter

What Is an OUTER JOIN?
The LEFT/RIGHT OUTER JOIN
The FULL OUTER JOIN
Uses for OUTER JOINs
Sample Statements
Summary
Problems for You to Solve

In the previous chapter, we covered all the "ins" of JOINs—linking two or more tables or result sets using INNER JOIN to find all the rows that match. Now it's time to talk about the "outs"—linking tables and finding out not only the rows that match but also the rows that don't match.

What Is an OUTER JOIN?

As we explained in the previous chapter, the SQL Standard defines several types of JOIN operations to link two or more tables or result sets. An OUTER JOIN asks your database system to return not only the rows that match on the criteria you specify but also the unmatched rows from either one or both of the two sets you want to link.

Let's suppose, for example, that you want to fetch information from the School Scheduling database about students and the classes for which they're registered. As you learned in the previous chapter, an INNER JOIN returns only students who have registered for a class and classes for which a student has registered. It won't return any students who have been accepted at the school but haven't signed up for any classes yet, nor will it return any classes that are on the schedule but for which no student has yet shown an interest.

What if you want to list *all* students and the classes for which they are registered, if any? Conversely, suppose you want a list of *all* the classes and the students who have registered for those classes, if any. To solve this sort of problem, you need to ask for an OUTER JOIN.

Figure 9–1 uses a set diagram to show one possible relationship between students and classes. As you can see, a few students haven't registered for a class yet, and a few classes do not yet have any students signed up to take the class.

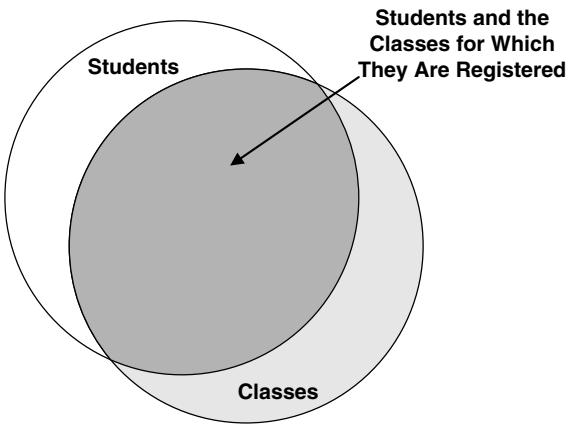


Figure 9–1 *A possible relationship between students and classes*

If you ask for *all* students and the classes for which they are registered, you'll get a result set resembling Figure 9–2.

You might ask, "What will I see for the students who haven't registered for any classes?" If you remember the concept of a Null or "nothing" value discussed in Chapter 5, Getting More Than Simple Columns, you know what you'll see: When you ask for all students joined with any classes, your database system will return a Null value in all columns from the Classes table

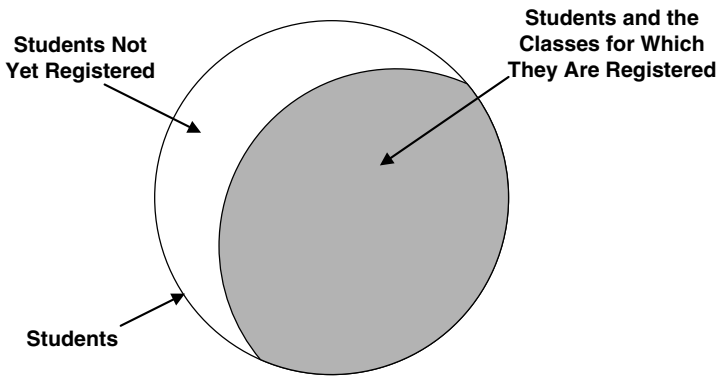


Figure 9–2 *All students and the classes for which they are registered*

when it finds a student who is not yet registered for any classes. If you think about the concept of a difference between two sets (discussed in Chapter 7, Thinking in Sets), the rows with a Null value in the columns from the Classes table represent the difference between the set of all students and the set of students who have registered for a class.

Likewise, if you ask for all classes and any students who registered for classes, the rows with Null values in the columns from the Students table represent the difference between the set of all classes and the set of classes for which students have registered. As we promised, using an OUTER JOIN with a test for Null values is an alternate way to discover the difference between two sets. Unlike a true EXCEPT operation that matches on entire rows from the two sets, you can specify the match in a JOIN operation on just a few specific columns (usually the primary key and the foreign key).

The LEFT/RIGHT OUTER JOIN

You'll generally use the OUTER JOIN form that asks for all the rows from one table or result set and any matching rows from a second table or result set. To do this, you specify either a LEFT OUTER JOIN or a RIGHT OUTER JOIN.

What's the difference between LEFT and RIGHT? Remember from the previous chapter that to specify an INNER JOIN on two tables, you name the first table, include the JOIN keyword, and then name the second table. When you begin building queries using OUTER JOIN, the SQL Standard considers the

first table you name as the one on the "left," and the second table as the one on the "right." So, if you want all the rows from the first table and any matching rows from the second table, you'll use a LEFT OUTER JOIN. Conversely, if you want all the rows from the second table and any matching rows from the first table, you'll specify a RIGHT OUTER JOIN.

Syntax

Let's examine the syntax needed to build either a LEFT or RIGHT OUTER JOIN.

Using Tables

We'll start simply with defining an OUTER JOIN using tables. Figure 9–3 shows the syntax diagram for creating a query with an OUTER JOIN on two tables.

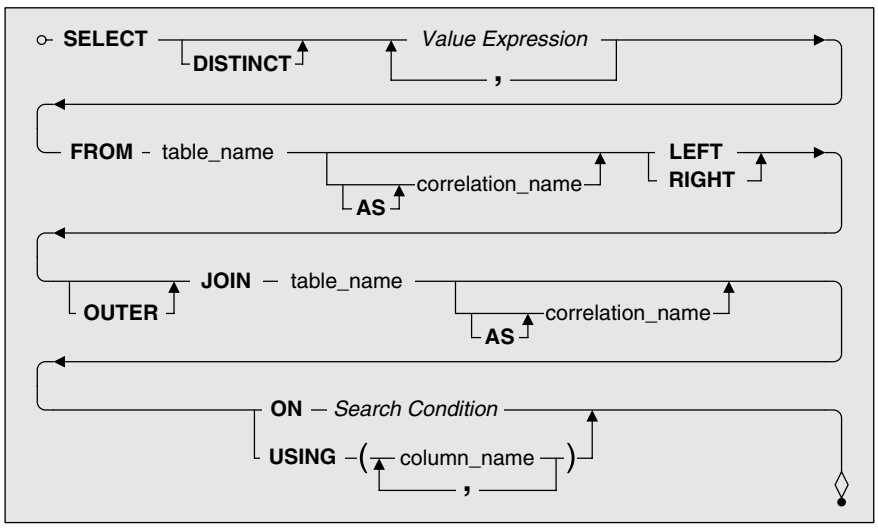


Figure 9–3 *Defining an OUTER JOIN on two tables*

Just like INNER JOIN (covered in Chapter 8), all the action happens in the FROM clause. (We left out the WHERE and ORDER BY clauses for now to simplify things.) Instead of specifying a single table name, you specify two table names and link them with the JOIN keyword. If you do not specify the type of JOIN you want, your database system assumes you want an INNER JOIN. In

this case, because you want an OUTER JOIN, you must explicitly state that you want either a LEFT JOIN or a RIGHT JOIN. The OUTER keyword is optional.

> ❖ **Note** For those of you following along with the complete syntax diagrams in Appendix A, SQL Standard Diagrams, note that we've pulled together the applicable parts (from *Select Statement, Table Reference,* and *Joined Table*) into simpler diagrams that explain the specific syntax we're discussing.

The critical part of any JOIN is the ON or USING clause that follows the second table and tells your database system how to perform the JOIN. To solve the JOIN, your database system logically combines every row in the first table with every row in the second table. (This combination of all rows from one table with all rows from a second table is called a *Cartesian product*.) It then applies the criteria in the ON or USING clause to find the matching rows to be returned. Because you asked for an OUTER JOIN, your database system also returns the unmatched rows from either the "left" or "right" table.

You learned about using a search condition to form a WHERE clause in Chapter 6, Filtering Your Data. You can use a search condition in the ON clause within a JOIN to specify a logical test that must be true in order to return any two linked rows. It only makes sense to write a search condition that compares at least one column from the first table with at least one column from the second table. Although you can write a very complex search condition, you can usually specify a simple equals comparison test on the primary key columns from one table with the foreign key columns from the other table.

To keep things simple, let's start with the same recipe classes and recipes example we used in the last chapter. Remember that in a well-designed database, you should break out complex classification names into a second table and then link the names back to the primary subject table via a simple key value. In the Recipes sample database, recipe classes appear in a table separate from recipes. Figure 9-4 shows the relationship between the Recipe_Classes and Recipes tables.

When you originally set up the kinds of recipes to save in your database, you might have started by entering all the recipe classes that came to mind. Now that you've entered a number of recipes, you might be interested in finding

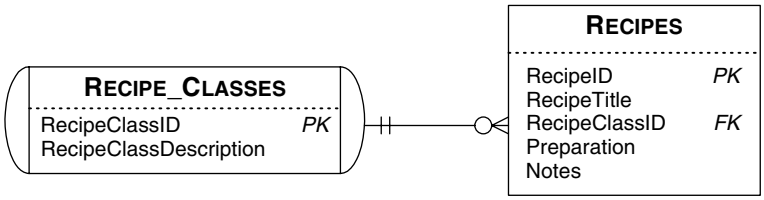


Figure 9–4 *Recipe classes are in a separate table from recipes.*

out which classes don't have any recipes entered yet. You might also be interested in listing *all* the recipe classes along with the names of recipes entered so far for each class. You can solve either problem with an OUTER JOIN.

❖ **Note** Throughout this chapter, we use the "Request/Translation/Clean Up/SQL" technique introduced in Chapter 4, Creating a Simple Query.

"Show me all the recipe types and any matching recipes in my database."

Translation: Select recipe class description and recipe title from the recipe classes table left outer joined with the recipes table on recipe class ID in the recipe classes table matching recipe class ID in the recipes table

Clean Up:
Select recipe class description ~~and~~ recipe title
from ~~the~~ recipe classes ~~table~~
left outer join~~ed with the~~ recipes ~~table~~
on recipe_ classes.recipe class ID ~~in the recipe classes table matching~~ = recipes.recipe class ID ~~in the recipes table~~

SQL:
```
SELECT Recipe_Classes.RecipeClassDescription,
   Recipes.RecipeTitle
FROM Recipe_Classes
LEFT OUTER JOIN Recipes
ON Recipe_Classes.RecipeClassID =
   Recipes.RecipeClassID
```

When using multiple tables in your FROM clause, remember to qualify fully each column name with the table name wherever you use it so that it's absolutely clear which column from which table you want. Note that we *had to* qualify the name of RecipeClassID in the ON clause because there are two columns named RecipeClassID—one in the Recipes table and one in the Recipe_Classes table.

❖ **Note** Although most commercial implementations of SQL support OUTER JOIN, some do not. If your database does not support OUTER JOIN, you can still solve the problem by listing all the tables you need in the FROM clause, then moving your search condition from the ON clause to the WHERE clause. You must consult your database documentation to learn the specific nonstandard syntax that your database requires to define the OUTER JOIN. For example, earlier versions of Microsoft SQL Server support this syntax. (Notice the asterisk in the WHERE clause.)

```
SELECT Recipe_Classes.RecipeClassDescription,
   Recipes.RecipeTitle
FROM Recipe_Classes, Recipes
WHERE Recipe_Classes.RecipeClassID *=
   Recipes.RecipeClassID
```

If you're using Oracle, the optional syntax is as follows. (Notice the plus sign in the WHERE clause.)

```
SELECT Recipe_Classes.RecipeClassDescription,
   Recipes.RecipeTitle
FROM Recipe_Classes, Recipes
WHERE Recipe_Classes.RecipeClassID =
   Recipes.RecipeClassID(+)
```

Quite frankly, these strange syntaxes were invented by database vendors that wanted to provide this feature long before a clearer syntax was defined in the SQL Standard. Thankfully, the SQL Standard syntax allows you to fully define the source for the final result set entirely within the FROM clause. Think of the FROM clause as fully defining a linked result set from which the database system obtains your answer. In the SQL Standard, you use the WHERE clause only to filter rows out of the result set defined by the FROM clause. Also, because the specific syntax for defining an OUTER JOIN via the WHERE clause varies by product, you might have to learn several different syntaxes if you work with multiple nonstandard products.

If you execute our example query in the Recipes sample database, you should see 16 rows returned. Because we didn't enter any soup recipes in the database, you'll get a Null value for RecipeTitle in the row where RecipeClassDescription is 'Soup'. To find only this one row, use this approach.

"List the recipe classes that do not yet have any recipes."

Translation Select recipe class description from the recipe classes table left outer joined with the recipes table on recipe class ID where recipe ID is empty

Clean Up Select recipe class description
from ~~the~~ recipe classes ~~table~~
left outer join~~ed with the~~ recipes ~~table~~
on recipe_classes.recipe class ID ~~in the recipes table matches~~
= recipes.recipe class ID ~~in the recipes table~~
where recipe ID is ~~empty~~ NULL

SQL
```
SELECT Recipe_Classes.RecipeClassDescription
FROM Recipe_Classes
LEFT OUTER JOIN Recipes
ON Recipe_Classes.RecipeClassID =
   Recipes.RecipeClassID
WHERE Recipes.RecipeID IS NULL
```

If you think about it, we've just done a difference or EXCEPT operation (see Chapter 7) using a JOIN. It's somewhat like saying, *"Show me all the recipe classes except the ones that already appear in the recipes table."* The set diagram in Figure 9–5 should help you visualize what's going on.

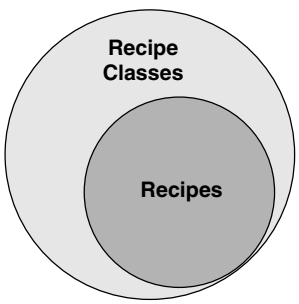


Figure 9–5 *A possible relationship between recipe classes and recipes*

In Figure 9–5, all recipes have a recipe class, but some recipe classes exist for which no recipe has yet been defined. When we add the IS NULL test, we're asking for all the rows in the lighter outer circle that don't have any matches in the set of recipes represented by the darker inner circle.

Notice that the diagram for an OUTER JOIN on tables in Figure 9–3 also has the optional USING clause. If the matching columns in the two tables have the same name and you want to join only on equal values, you can use the USING clause and list the column names. Let's do the previous problem again with USING.

"Display the recipe classes that do not yet have any recipes."

Translation	Select recipe class description from the recipe classes table left outer joined with the recipes table using recipe class ID where recipe ID is empty
Clean Up	Select recipe class description from ~~the~~ recipe classes ~~table~~ left outer join~~ed with the~~ recipes ~~table~~ using recipe class ID where recipe ID is ~~empty~~ NULL

SQL	`SELECT Recipe_Classes.RecipeClassDescription`
	`FROM Recipe_Classes`
	`LEFT OUTER JOIN Recipes`
	`USING (RecipeClassID)`
	`WHERE Recipes.RecipeID IS NULL`

The USING syntax is a lot simpler, isn't it? There's one small catch: Any column in the USING clause loses its table identity because the SQL Standard dictates that the database system must "coalesce" the two columns into a single column. In this example, there's only one RecipeClassID column as a result, so you can't reference Recipes.RecipeClassID or Recipe_Classes.RecipeClassID in the SELECT clause or any other clause.

Be aware that some database systems do not yet support USING. If you find that you can't use USING with your database, you can always get the same result with an ON clause and an equals comparison.

❖ **Note** The SQL Standard also defines a type of JOIN operation called a NATURAL JOIN. A NATURAL JOIN links the two specified tables by matching all the columns with the same name. If the only common columns are the linking columns and your database supports NATURAL JOIN, you can solve the example problem like this:

```
SELECT Recipe_Classes.RecipeClassDescription
FROM Recipe_Classes
NATURAL LEFT OUTER JOIN Recipes
WHERE Recipes.RecipeID IS NULL
```

Do not specify an ON or USING clause if you use the NATURAL keyword.

Embedding a SELECT Statement

As you recall from Chapter 8, most SQL implementations let you substitute an entire SELECT statement for any table name in your FROM clause. Of course, you must then assign a correlation name (see the section on assigning alias

names in Chapter 8) so that the result of evaluating your embedded query has a name. Figure 9–6 shows how to assemble an OUTER JOIN clause using embedded SELECT statements.

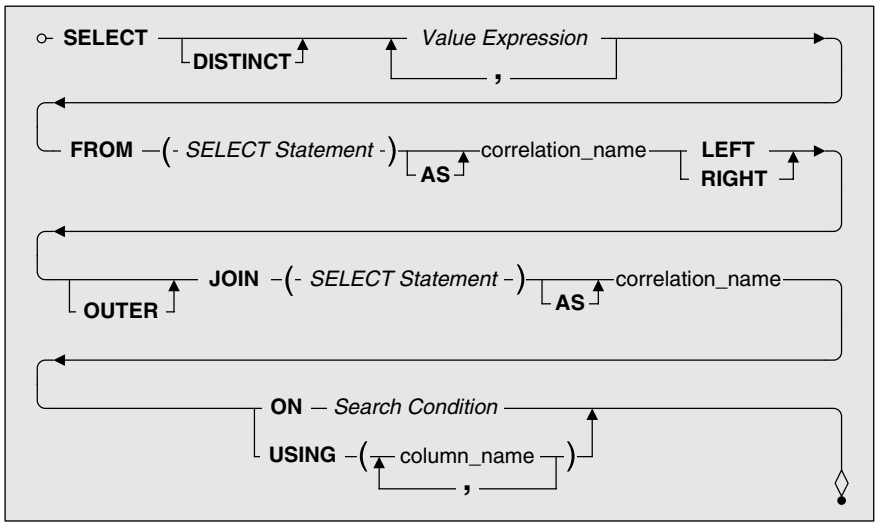


Figure 9–6 *An OUTER JOIN using SELECT statements*

Note that a SELECT statement can include all query clauses *except* an ORDER BY clause. Also, you can mix and match SELECT statements with table names on either side of the OUTER JOIN keywords.

Let's look at the Recipes and Recipe_Classes tables again. For this example, let's also assume that you are interested only in classes Salads, Soups, and Main courses. Here's the query with the Recipe_Classes table filtered in a SELECT statement that participates in a LEFT OUTER JOIN with the Recipes table.

```
SQL         SELECT RCFiltered.ClassName, R.RecipeTitle
            FROM
               (SELECT RecipeClassID,
                   RecipeClassDescription AS ClassName
                FROM Recipe_Classes AS RC
                WHERE RC.ClassName = 'Salads'
                   OR RC.ClassName = 'Soup'
                   OR RC.ClassName = 'Main Course')
              AS RCFiltered
            LEFT OUTER JOIN Recipes AS R
            ON RCFiltered.RecipeClassID = R.RecipeClassID
```

You must be careful when using a SELECT statement in a FROM clause. First, when you decide to substitute a SELECT statement for a table name, you must be sure to include not only the columns you want to appear in the final result but also any linking columns you need to perform the JOIN. That's why you see both RecipeClassID and RecipeClassDescription in the embedded statement. Just for fun, we gave RecipeClassDescription an alias name of ClassName in the embedded statement. As a result, the SELECT clause asks for ClassName rather than RecipeClassDescription. Note that the ON clause now references the correlation name (RCFiltered) of the embedded SELECT statement rather than the original name of the table or the correlation name we assigned the table inside the embedded SELECT statement.

As the query is stated for the actual Recipes sample database, you see one row with RecipeClassDescription of Soup with a Null value returned for RecipeTitle because there are no soup recipes in the sample database. We could just as easily have built a SELECT statement on the Recipes table on the right side of the OUTER JOIN. For example, we could have asked for recipes that contain the word "beef" in their titles, as in the following statement.

```
SQL         SELECT RCFiltered.ClassName, R.RecipeTitle
            FROM
               (SELECT RecipeClassID,
                   RecipeClassDescription AS ClassName
                FROM Recipe_Classes AS RC
                WHERE RC.ClassName = 'Salads'
                   OR RC.ClassName = 'Soup'
                   OR RC.ClassName = 'Main Course')
              AS RCFiltered
            LEFT OUTER JOIN
               (SELECT Recipes.RecipeClassID, Recipes.Recipe
                   Title
                FROM Recipes
                WHERE Recipes.RecipeTitle LIKE '%beef%')
              AS R
            ON RCFiltered.RecipeClassID = R.RecipeClassID
```

Keep in mind that the LEFT OUTER JOIN asks for *all* rows from the result set or table on the left side of the JOIN, regardless of whether any matching rows exist on the right side. The previous query not only returns a Soup row with a Null RecipeTitle (because there are no soups in the database at all) but also a Salad row with a Null. You might conclude that there are no salad recipes in the database. Actually, there *are* salads in the database but no salads with "beef" in the title of the recipe!

❖ **Note** You might have noticed that you can enter a full search condition as part of the ON clause in a JOIN. This is absolutely true, so it is perfectly legal in the SQL Standard to solve the example problem as follows.

```
SELECT Recipe_Classes.RecipeClassDescription,
   Recipes.RecipeTitle
FROM Recipe_Classes
LEFT OUTER JOIN Recipes
ON Recipe_Classes.RecipeClassID =
   Recipes.RecipeClassID
AND
   (Recipe_Classes.RecipeClassDescription = 'Salads'
OR Recipe_Classes.RecipeClassDescription = 'Soup'
OR Recipe_Classes.RecipeClassDescription =
   'Main Course')
AND Recipes.RecipeTitle LIKE '%beef%'
```

Unfortunately, we have discovered that some major implementations of SQL solve this problem incorrectly or do not accept this syntax at all! Therefore, we recommend that you always enter in the search condition in the ON clause only criteria that compare columns from the two tables or result sets. If you want to filter the rows from the underlying tables, do so with a separate search condition in a WHERE clause in an embedded SELECT statement.

Embedding JOINs within JOINs

Although you can solve many problems by linking just two tables, many times you'll need to link three, four, or more tables to get all the data to solve your request. For example, you might want to fetch all the relevant information about recipes—the type of recipe, the recipe name, and all the ingredients for the recipe—in one query. Now that you understand what you can do with an OUTER JOIN, you might also want to list *all* recipe classes—even those that have no recipes defined yet—and all the details about recipes and their ingredients. Figure 9–7 shows all the tables needed to answer this request.

Looks like you need data from *five* different tables! Just as in Chapter 8, you can do this by constructing a more complex FROM clause, embedding JOIN clauses within JOIN clauses. Here's the trick: Everywhere you can specify a table name, you can also specify an entire JOIN clause surrounded with parentheses. Figure 9–8 shows a simplified version of joining two tables. (We've left off the correlation name clauses and chosen the ON clause to form a simple INNER or OUTER JOIN of two tables.)

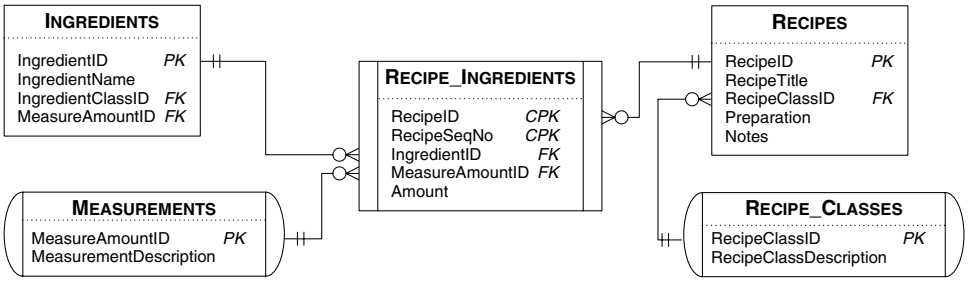


Figure 9–7 *The tables you need from the Recipes sample database to fetch all the information about recipes*

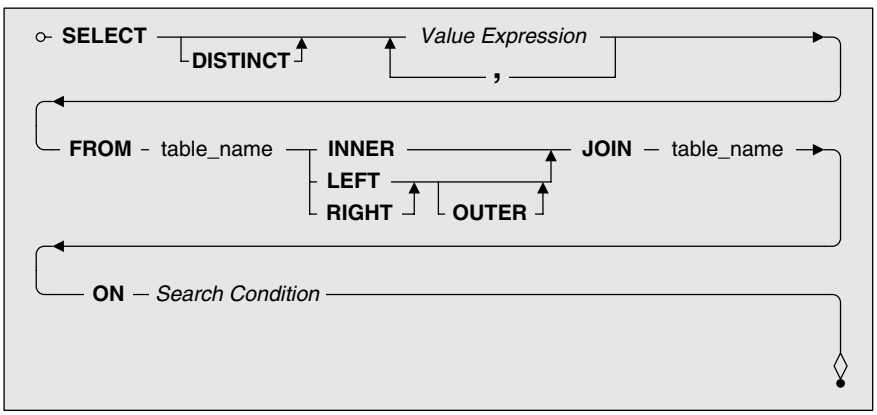


Figure 9–8 *A simple JOIN of two tables*

To add a third table to the mix, just place an open parenthesis before the first table name, add a close parenthesis after the search condition, and then insert another JOIN, a table name, the ON keyword, and another search condition. Figure 9–9 (on page 306) shows how to do this.

If you think about it, the JOIN of two tables inside the parentheses forms a logical table, or inner result set. This result set now takes the place of the first simple table name in Figure 9–8. You can continue this process of enclosing an entire JOIN clause in parentheses and then adding another JOIN keyword, table name, ON keyword, and search condition until you have all the result sets you need. Let's make a request that needs data from all the tables shown in Figure 9–7 and see how it turns out. (You might use this type of request for a report that lists all recipe types with details about the recipes in each type.)

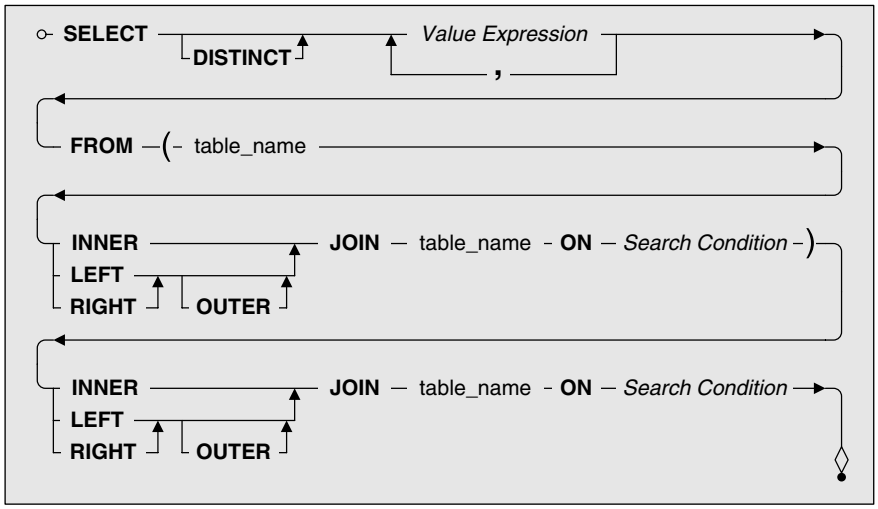


Figure 9–9 *A simple JOIN of three tables*

"I need all the recipe types, and then the matching recipe names, preparation instructions, ingredient names, ingredient step numbers, ingredient quantities, and ingredient measurements from my recipes database, sorted in recipe title and step number sequence."

Translation Select the recipe class description, recipe title, preparation instructions, ingredient name, recipe sequence number, amount, and measurement description from the recipe classes table left outer joined with the recipes table on recipe class ID in the recipe classes table matching recipe ID in the recipes table, then joined with the recipe ingredients table on recipe ID in the recipes table matching recipe ID in the recipe ingredients table, then joined with the ingredients table on ingredient ID in the ingredients table matching ingredient ID in the recipe ingredients table, and then finally joined with the measurements table on measurement amount ID in the measurements table matching measurement amount ID in the recipe ingredients table, order by recipe title and recipe sequence number

Clean Up Select ~~the~~ recipe class description, recipe title, preparation ~~instructions~~, ingredient name, recipe sequence number, amount, ~~and~~ measurement description
from ~~the~~ recipe classes ~~table~~
left outer join~~ed with the~~ recipes ~~table~~

on recipe_classes .recipe class ID ~~in the recipe classes table matching~~ = recipes .recipe class ID ~~in the recipes table,~~ ~~then~~ inner join~~ed with the~~ recipe ingredients ~~table~~
on recipes.recipe ID ~~in the recipes table matching~~
= recipe_ingredients.recipe ID ~~in the recipe ingredients table, then~~
inner join~~ed with the~~ ingredients ~~table~~
on ingredients.ingredient ID ~~in the ingredients table matching~~
= recipe_ingredients.ingredient ID
~~in the recipe ingredients table, and then~~
~~finally~~ inner join~~ed with the~~ measurements ~~table~~
on measurements.measurement amount ID ~~in the measurements table matching~~
= recipe_ingredients.measurement amount ID
~~in the recipe ingredients table,~~
order by recipe title, ~~and~~ recipe sequence number

SQL

```
SELECT Recipe_Classes.RecipeClassDescription,
   Recipes.RecipeTitle, Recipes.Preparation,
   Ingredients.IngredientName,
   Recipe_Ingredients.RecipeSeqNo,
   Recipe_Ingredients.Amount,
    Measurements.MeasurementDescription
FROM (((Recipe_Classes
LEFT OUTER JOIN Recipes
ON Recipe_Classes.RecipeClassID =
   Recipes.RecipeClassID)
INNER JOIN Recipe_Ingredients
ON Recipes.RecipeID =
    Recipe_Ingredients.RecipeID)
INNER JOIN Ingredients
ON Ingredients.IngredientID =
   Recipe_Ingredients.IngredientID)
INNER JOIN Measurements
ON Measurements.MeasureAmountID =
   Recipe_Ingredients.MeasureAmountID
ORDER BY RecipeTitle, RecipeSeqNo
```

In truth, you can substitute an entire JOIN of two tables anywhere you might otherwise place only a table name. In Figure 9–9, we implied that you must first join the first table with the second table and then join that result with the third table. You could also join the second and third tables first (as long as the third table is, in fact, related to the second table and not the first one) and then perform the final JOIN with the first table. Figure 9–10 (on page 308) shows you this alternate method.

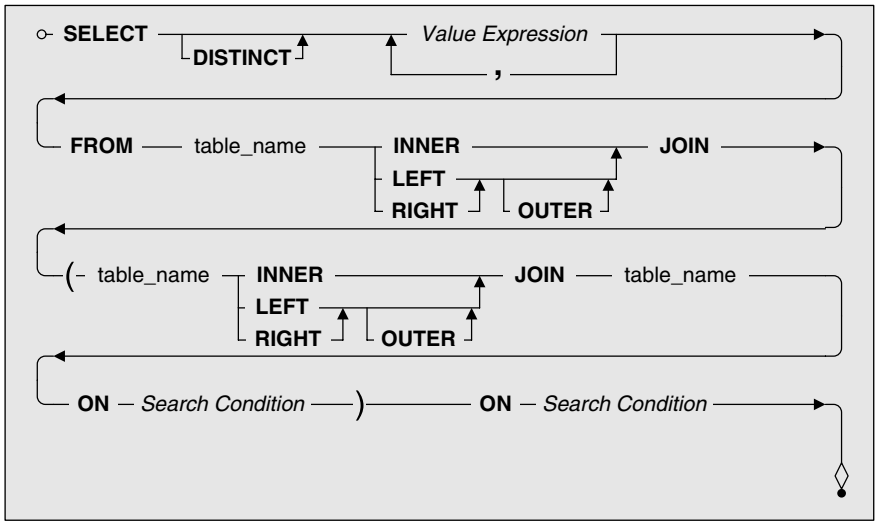


Figure 9–10 *Joining more than two tables in an alternate sequence*

To solve the request we just showed you using five tables, we could have also stated the SQL as follows.

```
SQL         SELECT Recipe_Classes.RecipeClassDescription,
               Recipes.RecipeTitle, Recipes.Preparation,
               Ingredients.IngredientName,
               Recipe_Ingredients.RecipeSeqNo,
               Recipe_Ingredients.Amount,
               Measurements.MeasurementDescription
            FROM Recipe_Classes
            LEFT OUTER JOIN
               (((Recipes
               INNER JOIN Recipe_Ingredients
               ON Recipes.RecipeID = Recipe_Ingredients.RecipeID)
               INNER JOIN Ingredients
               ON Ingredients.IngredientID =
                  Recipe_Ingredients.IngredientID)
               INNER JOIN Measurements
               ON Measurements.MeasureAmountID =
                  Recipe_Ingredients.MeasureAmountID)
            ON Recipe_Classes.RecipeClassID =
               Recipes.RecipeClassID
            ORDER BY RecipeTitle, RecipeSeqNo
```

Remember that the optimizers in some database systems are sensitive to the sequence of the JOIN definitions. If your query with many JOINs is taking a

long time to execute on a large database, it might run faster if you change the sequence of JOINs in your SQL statement.

You might have noticed that we used only one OUTER JOIN in the previous multiple-JOIN examples. You're probably wondering whether it's possible or even makes sense to use more than one OUTER JOIN in a complex JOIN. Let's assume that there are not only some recipe classes that don't have matching recipe rows but also some recipes that don't have any ingredients defined yet. In the previous example, you won't see any rows from the Recipes table that do not have any matching rows in the Recipe_Ingredients table because the INNER JOIN eliminates them. Let's ask for all recipes as well.

"I need all the recipe types, and then all the recipe names and preparation instructions, and then any matching ingredient names, ingredient step numbers, ingredient quantities, and ingredient measurements from my recipes database, sorted in recipe title and step number sequence."

Translation: Select the recipe class description, recipe title, preparation instructions, ingredient name, recipe sequence number, amount, and measurement description from the recipe classes table left outer joined with the recipes table on recipe class ID in the recipe classes table matching recipe class ID in the recipes table, then left outer joined with the recipe ingredients table on recipe ID in the recipes table matching recipe ID in the recipe ingredients table, then joined with the ingredients table on ingredient ID in the ingredients table matching ingredient ID in the recipe ingredients table, and then finally joined with the measurements table on measurement amount ID in the measurements table matching measurement amount ID in the recipe ingredients table, order by recipe title and recipe sequence number

Clean Up: Select ~~the~~ recipe class description, recipe title, preparation ~~instructions~~, ingredient name, recipe sequence number, amount, ~~and~~ measurement description
from ~~the~~ recipe classes ~~table~~
left outer join~~ed with the~~ recipes ~~table~~
on recipe_classes.recipe class ID ~~in the recipe classes table matching~~ = recipes.recipe class ID ~~in the recipes table, then~~ left outer join~~ed with the~~ recipe ingredients ~~table~~

on recipes.recipe ID ~~in the recipes table matching~~
= recipe_ingredients.recipe ID ~~in the recipe ingredients table, then~~
inner join~~ed with the~~ ingredients ~~table~~
on ingredients.ingredient ID ~~in the ingredients table matching~~
= recipe_ingredients.ingredient ID ~~in the recipe ingredients table, and then finally~~ inner join~~ed with the~~ measurements ~~table~~
on measurement.measurement amount ID
~~in the measurements table matching~~
= recipe_ingredients.measurement amount ID
~~in the recipe ingredients table,~~
order by recipe title ~~and~~ recipe sequence number

SQL

```
SELECT Recipe_Classes.RecipeClassDescription,
   Recipes.RecipeTitle, Recipes.Preparation,
   Ingredients.IngredientName,
   Recipe_Ingredients.RecipeSeqNo,
   Recipe_Ingredients.Amount,
   Measurements.MeasurementDescription
FROM (((Recipe_Classes
LEFT OUTER JOIN Recipes
ON Recipe_Classes.RecipeClassID =
   Recipes.RecipeClassID)
LEFT OUTER JOIN Recipe_Ingredients
ON Recipes.RecipeID =
   Recipe_Ingredients.RecipeID)
INNER JOIN Ingredients
ON Ingredients.IngredientID =
   Recipe_Ingredients.IngredientID)
INNER JOIN Measurements
ON Measurements.MeasureAmountID =
   Recipe_Ingredients.MeasureAmountID
ORDER BY RecipeTitle, RecipeSeqNo
```

Be careful! This sort of multiple OUTER JOIN works as expected only if you're following a path of one-to-many relationships. Let's look at the relationships between Recipe_Classes, Recipes, and Recipe_Ingredients again, as shown in Figure 9–11.

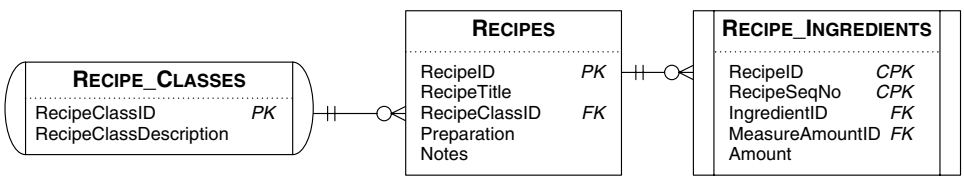


Figure 9–11 *The relationships between the Recipe_Classes, Recipes, and Recipe_Ingredients tables*

You might see a one-to-many relationship sometimes called a *parent-child relationship*. Each parent row (on the "one" side of the relationship) might have zero or more children rows (on the "many" side of the relationship). Unless you have orphaned rows on the "many" side (for example, a row in Recipes that has a Null in its RecipeClassID column), *every* row in the child table should have a matching row in the parent table. So it makes sense to say `Recipe_Classes LEFT JOIN Recipes` to pick up any parent rows in Recipe_Classes that don't have any children yet in Recipes. `Recipe_Classes RIGHT JOIN Recipes` should (barring any orphaned rows) give you the same result as an INNER JOIN.

Likewise, it makes sense to ask for `Recipes LEFT JOIN Recipe_Ingredients` because you might have some recipes for which no ingredients have yet been entered. `Recipes RIGHT JOIN Recipe_Ingredients` doesn't work because the linking column (RecipeID) in Recipe_Ingredients is also part of that table's compound primary key. Therefore, you are guaranteed to have no orphaned rows in Recipe_Ingredients because no column in a primary key can contain a Null value.

Now, let's take it one step further and ask for all ingredients, including those not yet included in any recipes. First, take a close look at the relationships between the tables, including the Ingredients table, as shown in Figure 9–12.

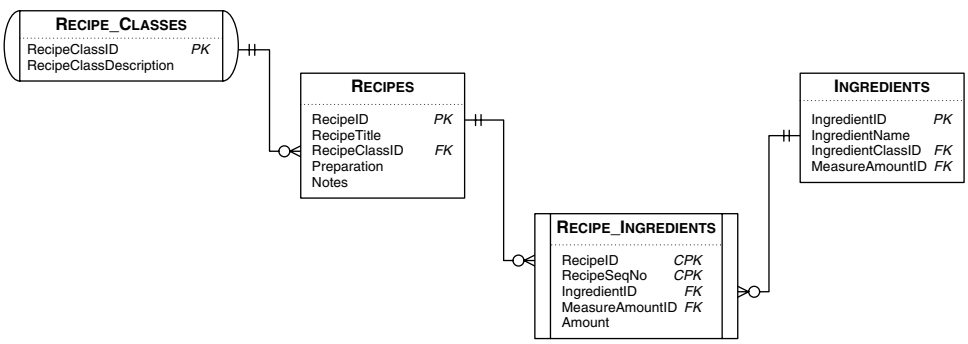


Figure 9–12 *The relationships between the Recipe_Classes, Recipes, Recipe_Ingredients, and Ingredients tables*

Let's try this request. (Caution: There's a trap here!)

> *"I need all the recipe types, and then all the recipe names and preparation instructions, and then any matching ingredient step numbers, ingredient quantities, and ingredient measurements, and finally all ingredient names from my recipes database, sorted in recipe title and step number sequence."*

Translation Select the recipe class description, recipe title, preparation instructions, ingredient name, recipe sequence number, amount, and measurement description from the recipe classes table left outer joined with the recipes table on recipe class ID in the recipe classes table matches class ID in the recipes table, then left outer joined with the recipe ingredients table on recipe ID in the recipes table matches recipe ID in the recipe ingredients table, then joined with the measurements table on measurement amount ID in the measurements table matches measurement amount ID in the measurements table, and then finally right outer joined with the ingredients table on ingredient ID in the ingredients table matches ingredient ID in the recipe ingredients table, order by recipe title and recipe sequence number

Clean Up Select ~~the~~ recipe class description, recipe title, preparation ~~instructions~~, ingredient name, recipe sequence number, amount, ~~and~~ measurement description
from ~~the~~ recipe classes ~~table~~ left outer join~~ed with the~~ recipes ~~table~~
on recipe_classes .recipe class ID ~~in the recipe classes table matches~~
= recipes .class ID ~~in the recipes table,~~
~~then~~ left outer join~~ed with the~~ recipe ingredients ~~table~~
on recipes.recipe ID ~~in the recipes table matches~~
= recipe_ingredients.recipe ID ~~in the recipe ingredients table,~~
~~then~~ inner join~~ed with the~~ measurements ~~table~~
on measurements.measurement amount ID
~~in the measurements table matches~~
= measurements.measurement amount ID
~~in the measurements table,~~
~~and then finally~~ right outer join~~ed with the~~ ingredients ~~table~~
on ingredients.ingredient ID ~~in the ingredients table matches~~
= recipe_ingredients.ingredient ID ~~in the recipe ingredients table,~~
order by recipe title, ~~and~~ recipe sequence number

SQL

```
SELECT Recipe_Classes.RecipeClassDescription,
   Recipes.RecipeTitle, Recipes.Preparation,
   Ingredients.IngredientName,
   Recipe_Ingredients.RecipeSeqNo,
   Recipe_Ingredients.Amount,
   Measurements.MeasurementDescription
FROM (((Recipe_Classes
LEFT OUTER JOIN Recipes
ON Recipe_Classes.RecipeClassID =
   Recipes.RecipeClassID)
LEFT OUTER JOIN Recipe_Ingredients
ON Recipes.RecipeID =
   Recipe_Ingredients.RecipeID)
INNER JOIN Measurements
ON Measurements.MeasureAmountID =
   Recipe_Ingredients.MeasureAmountID)
```

```
RIGHT OUTER JOIN Ingredients
ON Ingredients.IngredientID =
   Recipe_Ingredients.IngredientID
ORDER BY RecipeTitle, RecipeSeqNo
```

Do you think this will work? Actually, the answer is a resounding NO! Most database systems analyze the entire FROM clause and then try to determine the most efficient way to assemble the table links. Let's assume, however, that the database decides to fully honor how we've grouped the JOINs within parentheses. This means that the database system will work from the innermost JOIN first (Recipe_Classes joined with Recipes) and then work outward.

Because some rows in Recipe_Classes might not have any matching rows in Recipes, this first JOIN returns rows that have a Null value in RecipeClassID. Looking back at Figure 9–12, you can see that there's a one-to-many relationship between Recipe_Classes and Recipes. Unless some recipes exist that haven't been assigned a recipe class, we should get *all* the rows from the Recipes table anyway! The next JOIN with the Recipe_Ingredients table also asks for a LEFT OUTER JOIN. We want all the rows, regardless of any Null values, from the previous JOIN (of Recipe_Classes with Recipes) and any matching rows in Recipe_Ingredients. Again, because some rows in Recipe_Classes might not have matching rows in Recipes or some rows in Recipes might not have matching rows in Recipe_Ingredients, several of the rows might have a Null in the IngredientID column from the Recipe_Ingredients table. What we're doing with both JOINs is "walking down" the one-to-many relationships from Recipe_Classes to Recipes and then from Recipes to Recipe_Ingredients. So far, so good. (By the way, the final INNER JOIN with Measurements is inconsequential—we know that all Ingredients have a valid MeasureAmountID.)

Now the trouble starts. The final RIGHT OUTER JOIN asks for all the rows from Ingredients and *any matching* rows from the result of the previous JOINs. Remember from Chapter 5 that a Null is a very special value—it cannot be equal to any other value, not even another Null. When we ask for *all* the rows in Ingredients, the IngredientID in all these rows has a non-Null value. None of the rows from the previous JOIN that have a Null in IngredientID will match at all, so the final JOIN throws them away! You will see any ingredient that isn't used yet in any recipe, but you won't see recipe classes that have no recipes or recipes that have no ingredients.

If your database system decides to solve the query by performing the JOINs in a different order, you might see recipe classes that have no recipes and recipes that have no ingredients, but you won't see ingredients not yet used in any

recipe because of the Null matching problem. Some database systems might recognize this logic problem and refuse to solve your query at all—you'll see something like an "ambiguous OUTER JOINs" error message. The problem we're now experiencing results from trying to "walk back up" a many-to-one relationship with an OUTER JOIN going in the other direction. Walking down the hill is easy, but walking back up the other side requires special tools. What's the solution to this problem? Read on to the next section to find out!

The FULL OUTER JOIN

A FULL OUTER JOIN is neither "left" nor "right"—it's both! It includes *all* the rows from both of the tables or result sets participating in the JOIN. When no matching rows exist for rows on the "left" side of the JOIN, you see Null values from the result set on the "right." Conversely, when no matching rows exist for rows on the "right" side of the JOIN, you see Null values from the result set on the "left."

Syntax

Now that you've been working with JOINs for a while, the syntax for a FULL OUTER JOIN should be pretty obvious. You can study the syntax diagram for a FULL OUTER JOIN in Figure 9–13.

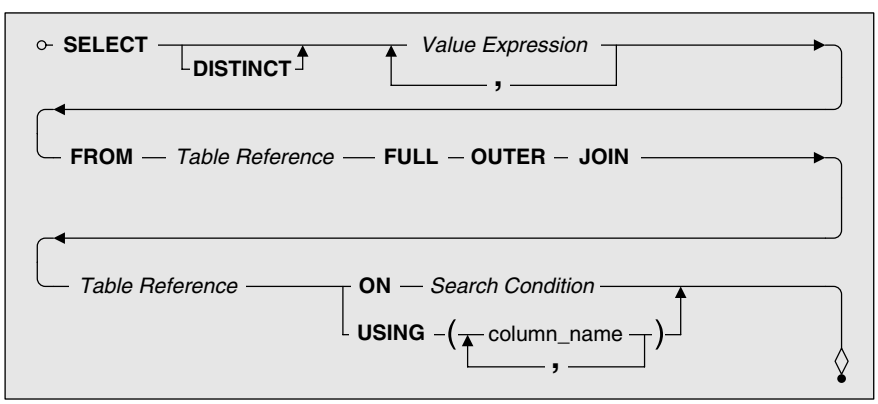


Figure 9–13 *The syntax diagram for a FULL OUTER JOIN*

To simplify things, we're now using the term *table reference* in place of a table name, a SELECT statement, or the result of another JOIN. Let's take another look at the problem we introduced at the end of the previous section. We can now solve it properly using a FULL OUTER JOIN.

"I need all the recipe types, and then all the recipe names and preparation instructions, and then any matching ingredient step numbers, ingredient quantities, and ingredient measurements, and finally all ingredient names from my recipes database, sorted in recipe title and step number sequence."

Translation Select the recipe class description, recipe title, preparation instructions, ingredient name, recipe sequence number, amount, and measurement description from the recipe classes table full outer joined with the recipes table on recipe class ID in the recipe classes table matches recipe class ID in the recipes table, then left outer joined with the recipe ingredients table on recipe ID in the recipes table matches recipe ID in the recipe ingredients table, then joined with the measurements table on measurement amount ID in the measurements table matches measurement amount ID in the recipe ingredients table, and then finally full outer joined with the ingredients table on ingredient ID in the ingredients table matches ingredient ID in the recipe ingredients table, order by recipe title and recipe sequence number

Clean Up Select ~~the~~ recipe class description, recipe title, preparation ~~instructions~~, ingredient name, recipe sequence number, amount, ~~and~~ measurement description
from ~~the~~ recipe classes ~~table~~
full outer join~~ed with the~~ recipes ~~table~~
on recipe_classes .recipe class ID ~~in the recipe classes table matches~~
= recipes .recipe class ID ~~in the recipes table,~~
~~then~~ left outer join~~ed with the~~ recipe ingredients ~~table~~
on recipes.recipe ID ~~in the recipes table matches~~
= recipe_ingredients.recipe ID ~~in the recipe ingredients table,~~
~~then~~ inner join~~ed with the~~ measurements ~~table~~
on measurements.measurement amount ID
~~in the measurements table matches~~
= recipe_ingredients.measurement amount ID
~~in the recipe ingredients table,~~
~~and then finally~~ full outer join~~ed with the~~ ingredients ~~table~~
on ingredients.ingredient ID ~~in the ingredients table matches~~
= recipe_ingredients .ingredient ID ~~in the recipe ingredients table,~~
order by recipe title ~~and~~ recipe sequence number

SQL
```
SELECT Recipe_Classes.RecipeClassDescription,
   Recipes.RecipeTitle, Recipes.Preparation,
   Ingredients.IngredientName,
   Recipe_Ingredients.RecipeSeqNo,
```

```
   Recipe_Ingredients.Amount,
   Measurements.MeasurementDescription
FROM (((Recipe_Classes
FULL OUTER JOIN Recipes
  ON Recipe_Classes.RecipeClassID =
     Recipes.RecipeClassID)
  LEFT OUTER JOIN Recipe_Ingredients
  ON Recipes.RecipeID =
     Recipe_Ingredients.RecipeID)
  INNER JOIN Measurements
  ON Measurements.MeasureAmountID =
     Recipe_Ingredients.MeasureAmountID)
  FULL OUTER JOIN Ingredients
  ON Ingredients.IngredientID =
     Recipe_Ingredients.IngredientID
ORDER BY RecipeTitle, RecipeSeqNo
```

The first and last JOINs now ask for *all* rows from both sides of the JOIN, so the problem with Nulls not matching is solved. You should now see not only recipe classes for which there are no recipes and recipes for which there are no ingredients but also ingredients that haven't been used in a recipe yet. You might get away with using a LEFT OUTER JOIN for the first JOIN, but because you can't predict in advance how your database system decides to nest the JOINs, you should ask for a FULL OUTER JOIN on both ends to ensure the right answer.

❖ **Note** As you might expect, database systems that do not support the SQL Standard syntax for LEFT OUTER JOIN or RIGHT OUTER JOIN also have a special syntax for FULL OUTER JOIN. You must consult your database documentation to learn the specific nonstandard syntax that your database requires to define the OUTER JOIN. For example, earlier versions of Microsoft SQL Server support the following syntax. (Notice the asterisks in the WHERE clause.)

```
SELECT Recipe_Classes.RecipeClassDescription,
   Recipes.RecipeTitle
FROM Recipe_Classes, Recipes
WHERE Recipe_Classes.RecipeClassID *=*
   Recipes.RecipeClassID
```

Products that do not support any FULL OUTER JOIN syntax but do support LEFT or RIGHT OUTER JOIN yield an equivalent result by performing a UNION on a LEFT and RIGHT OUTER JOIN. We'll discuss UNION in more detail in the next chapter. Because the specific syntax for defining a FULL OUTER JOIN using the WHERE clause varies by product, you might have to learn several different syntaxes if you work with multiple nonstandard products.

FULL OUTER JOIN on Non-Key Values

Thus far, we have been discussing using OUTER JOINs to link tables or result sets on related key values. You can, however, solve some interesting problems by using an OUTER JOIN on non-key values. For example, the previous chapter showed how to find students and staff who have the same first name in the School Scheduling database. Suppose you're interested in listing *all* staff members and *all* students and showing the ones who have the same first name as well. You can do that with a FULL OUTER JOIN.

"Show me all the students and all the teachers and list together those who have the same first name."

Translation
Select student full name and staff full name from the students table full outer joined with the staff table
on first name in the students table matches
first name in the staff table

Clean Up
Select student full name ~~and~~ staff full name
from ~~the~~ students ~~table~~
full outer join~~ed with the~~ staff ~~table~~
on students.first name ~~in the students table matches~~
= staff.first name ~~in the staff table~~

SQL

```
SELECT (Students.StudFirstName || ' ' ||
   Students.StudLastName) AS StudFullName,
   (Staff.StfFirstName || ' ' ||
   Staff.StfLastName) AS StfFullName
FROM Students
FULL OUTER JOIN Staff
ON Students.StudFirstName =
   Staff.StfFirstName
```

UNION JOIN

No discussion of OUTER JOINs would be complete without at least an honorable mention to UNION JOIN. In the SQL Standard, a UNION JOIN is a FULL OUTER JOIN with the matching rows removed. Figure 9–14 (on page 318) shows the syntax.

As you might expect, not many commercial implementations support a UNION JOIN. Quite frankly, we're hard pressed to think of a good reason why you would want to do a UNION JOIN.

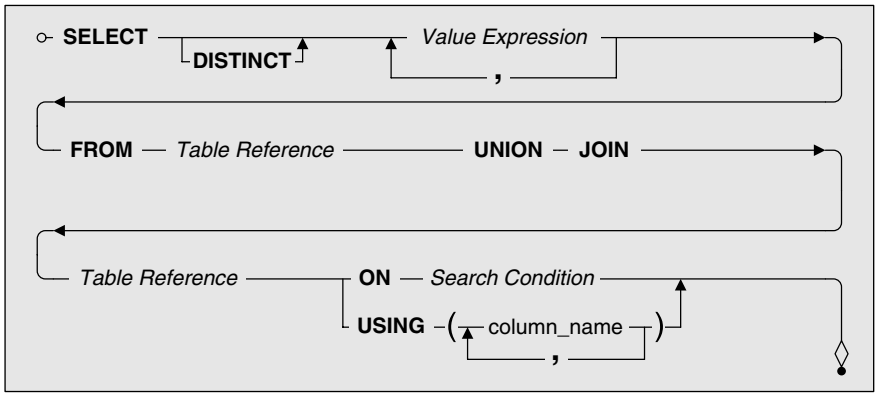


Figure 9–14 *The SQL syntax for a UNION JOIN*

Uses for OUTER JOINs

Because an OUTER JOIN lets you see not only the matched rows but also the unmatched ones, it's great for finding out which, if any, rows in one table do not have a matching related row in another table. It also helps you find rows that have matches on a few rows but not on all. In addition, it's useful for creating input to a report where you want to show all categories (regardless of whether matching rows exist in other tables) or all customers (regardless of whether a customer has placed an order). Following is a small sample of the kinds of requests you can solve with an OUTER JOIN.

Find Missing Values

Sometimes you just want to find what's missing. You do so by using an OUTER JOIN with a test for Null. Here are some "missing value" problems you can solve.

"What products have never been ordered?"
"Show me customers who have never ordered a helmet."
"List entertainers who have never been booked."
"Display agents who haven't booked an entertainer."
"Show me tournaments that haven't been played yet."
"List the faculty members not teaching a class."
"Display students who have never withdrawn from a class."
"Show me classes that have no students enrolled."

"List ingredients not used in any recipe yet."

"Display missing types of recipes."

Find Partially Matched Information

Particularly for reports, it's useful to be able to list all the rows from one or more tables along with any matching rows from related tables. Here's a sample of "partially matched" problems you can solve with an OUTER JOIN.

"List all products and the dates for any orders."

"Display all customers and any orders for bicycles."

"Show me all entertainment styles and the customers who prefer those styles."

"List all entertainers and any engagements they have booked."

"List all bowlers and any games they bowled over 160."

"Display all tournaments and any matches that have been played."

"Show me all subject categories and any classes for all subjects."

"List all students and the classes for which they are currently enrolled."

"Display all faculty and the classes they are scheduled to teach."

"List all recipe types, all recipes, and any ingredients involved."

"Show me all ingredients and any recipes they're used in."

Sample Statements

You now know the mechanics of constructing queries using OUTER JOIN and have seen some of the types of requests you can answer with an OUTER JOIN. Let's look at a fairly robust set of samples, all of which use OUTER JOIN. These examples come from each of the sample databases, and they illustrate the use of the OUTER JOIN to find either missing values or partially matched values.

We've also included sample result sets that would be returned by these operations and placed them immediately after the SQL syntax line. The name that appears immediately above a result set is the name we gave each query in the sample data on the companion CD you'll find bound into the back of the book. We stored each query in the appropriate sample database (as indicated within the example) and prefixed the names of the queries relevant to this chapter with "CH09." You can follow the instructions in the Introduction of this book to load the samples onto your computer and try them.

❖ **Note** Because many of these examples use complex JOINs, the optimizer for your database system might choose a different way to solve these queries. For this reason, the first few rows might not exactly match the result you obtain, but the total number of rows should be the same. To simplify the process, we have combined the Translation and Clean Up steps for all the following examples.

Sales Orders Database

"What products have never been ordered?"

Translation/ Clean Up: Select product number ~~and~~ product name from ~~the~~ products ~~table~~ left outer join~~ed with the~~ order details ~~table~~ on products.product number ~~in the products table matches~~ = order_details .product number ~~in the order details table~~ where ~~the~~ order detail order number is null

SQL:

```
SELECT Products.ProductNumber,
   Products.ProductName
FROM Products LEFT OUTER JOIN Order_Details
ON Products.ProductNumber =
   Order_Details.ProductNumber
WHERE Order_Details.OrderNumber IS NULL
```

CH09_Products_Never_Ordered (2 rows)

ProductNumber	ProductName
4	Victoria Pro All Weather Tires
23	Ultra-Pro Rain Jacket

"Display all customers and any orders for bicycles."

Translation 1: Select customer full name, order date, product name, quantity ordered, and quoted price from the customers table left outer joined with the orders table on customer ID, then joined with the order details table on order number, then joined with the products table on product number, then finally joined with the categories table on category ID where category description is 'Bikes'

Translation 2/ Clean Up: Select customer full name, order date, product name, quantity ordered, ~~and~~ quoted price from ~~the~~ customers ~~table~~ left outer join~~ed with~~

(Select customer ID, order date, product name,
quantity ordered, ~~and~~ quoted price
from ~~the~~ orders ~~table~~
inner join~~ed with the~~ order details ~~table~~
on orders .order number ~~in the orders table matches~~
= order_details .order number ~~in the order details table,~~
~~then~~ join~~ed with the~~ products ~~table~~
on order_details.product number ~~in the order details table matches~~ = products.product number ~~in the products table, then finally~~ join~~ed with the~~ categories ~~table~~
on categories.category ID ~~in the categories table matches~~
= products.category ID ~~in the products table~~
where category description ~~is~~ = 'Bikes') as rd
on customers.customer ID in the customers table matches
= rd.customerID ~~in the embedded SELECT statement~~

❖ **Note** Because we're looking for specific orders (bicycles), we split the translation process into two steps to show that the orders need to be filtered before applying an OUTER JOIN.

```
SQL         SELECT Customers.CustFirstName || ' ' ||
               Customers.CustLastName AS CustFullName,
               RD.OrderDate, RD.ProductName,
               RD.QuantityOrdered, RD.QuotedPrice
            FROM Customers
            LEFT OUTER JOIN
              (SELECT Orders.CustomerID, Orders.OrderDate,
                  Products.ProductName,
                  Order_Details.QuantityOrdered,
                  Order_Details.QuotedPrice
                FROM ((Orders
                INNER JOIN Order_Details
                ON Orders.OrderNumber =
                   Order_Details.OrderNumber)
                INNER JOIN Products
                ON Order_Details.ProductNumber =
                   Products.ProductNumber)
                INNER JOIN Categories
                ON Categories.CategoryID =
                   Products.CategoryID
                WHERE Categories.CategoryDescription =
                   'Bikes')
              AS RD
            ON Customers.CustomerID = RD.CustomerID
```

❖ **Note** This request is really tricky because you want to list *all* customers OUTER JOINed with only the orders for bikes. If you turn Translation 1 directly into SQL, you won't find any of the customers who have not ordered a bike! An OUTER JOIN from Customers to Orders *will* return all customers and any orders. When you add the filter to select only bike orders, that's all you will get—customers who ordered bikes.

Translation 2 shows you how to do it correctly—create an inner result set that returns only orders for bikes, and then OUTER JOIN that with Customers to get the final answer.

CH09_All_Customers_And_Any_Bike_Orders (913 rows)

CustFullName	OrderDate	ProductName	QuantityOrdered	QuotedPrice
Suzanne Viescas				
William Thompson	2007-12-23	Trek 9000 Mountain Bike	5	$1,164.00
William Thompson	2008-01-15	Trek 9000 Mountain Bike	6	$1,164.00
William Thompson	2007-10-11	Viscount Mountain Bike	2	$635.00
William Thompson	2007-10-05	Viscount Mountain Bike	5	$615.95
William Thompson	2008-01-15	Trek 9000 Mountain Bike	4	$1,200.00
William Thompson	2007-10-11	Trek 9000 Mountain Bike	3	$1,200.00
William Thompson	2008-01-07	Trek 9000 Mountain Bike	2	$1,200.00
<< more rows here >>				

(Looks like William Thompson is a really good customer!)

Entertainment Agency Database

"List entertainers who have never been booked."

Translation/ Clean Up

Select entertainer ID ~~and~~ entertainer stage name
from ~~the~~ entertainers ~~table~~
left outer join~~ed with the~~ engagements ~~table~~
on entertainers.entertainer ID ~~in the entertainers table matches~~
= engagements.entertainer ID ~~in the engagements table~~
where engagement number is null

SQL

```
SELECT Entertainers.EntertainerID,
   Entertainers.EntStageName
FROM Entertainers
LEFT OUTER JOIN Engagements
ON Entertainers.EntertainerID =
   Engagements.EntertainerID
WHERE Engagements.EngagementNumber IS NULL
```

CH09_Entertainers_Never_Booked (1 row)

EntertainerID	EntStageName
1009	Katherine Ehrlich

"Show me all musical styles and the customers who prefer those styles."

Translation/ Clean Up

Select style ID, style name, customer ID, customer first name, ~~and~~ customer last name
from ~~the~~ musical styles ~~table~~
left outer join~~ed with~~
(~~the~~ musical preferences ~~table~~ inner join~~ed with the~~ customers ~~table~~
on musical_preferences.customer ID ~~in the musical preferences table matches~~
= customers.customer ID ~~in the customers table~~)
on musical_styles.style ID ~~in the musical styles table matches~~
= musical_preferences.style ID ~~in the musical preferences table~~

SQL

```
SELECT Musical_Styles.StyleID,
   Musical_Styles.StyleName,
   Customers.CustomerID,
   Customers.CustFirstName,
   Customers.CustLastName
FROM Musical_Styles
LEFT OUTER JOIN (Musical_Preferences
   INNER JOIN Customers
   ON Musical_Preferences.CustomerID =
      Customers.CustomerID)
ON Musical_Styles.StyleID =
   Musical_Preferences.StyleID
```

CH09_All_Styles_And_Any_Customers (41 rows)

StyleID	StyleName	CustomerID	CustFirstName	CustLastName
1	40s Ballroom Music	10015	Carol	Viescas
1	40s Ballroom Music	10011	Joyce	Bonnicksen
2	50s Music			
3	60s Music	10002	Deb	Waldal
4	70s Music	10007	Liz	Keyser
5	80s Music	10014	Mark	Rosales
6	Country	10009	Sarah	Thompson
7	Classical	10005	Elizabeth	Hallmark
<< *more rows here* >>				

(Looks like nobody likes 50s music!)

❖ **Note** We very carefully phrased the FROM clause to influence the database system to first perform the INNER JOIN between Musical_Preferences and Customers, and then OUTER JOINed that with Musical_Styles. If your database tends to process JOINs from left to right, you might have to state the FROM clause with the INNER JOIN first followed by a RIGHT OUTER JOIN to Musical_Styles. In Microsoft Office Access, we had to state the INNER JOIN as an embedded SELECT statement to get it to return the correct answer.

School Scheduling Database

"List the faculty members not teaching a class."

Translation/ Clean Up: Select staff first name ~~and~~ staff last name
from ~~the~~ staff ~~table~~ left outer join~~ed with the~~ faculty classes ~~table~~
on staff.staff ID ~~in the staff table matches~~
= faculty_classes.staff ID ~~in the faculty classes table~~
where class ID is null

SQL:

```
SELECT Staff.StfFirstName, Staff.StfLastName,
FROM Staff
LEFT OUTER JOIN Faculty_Classes
ON Staff.StaffID = Faculty_Classes.StaffID
WHERE Faculty_Classes.ClassID IS NULL
```

CH09_Staff_Not_Teaching (4 rows)

StfFirstName	StfLastName
Jeffrey	Smith
Tim	Smith
Kathryn	Patterson
Joe	Rosales III

"Display students who have never withdrawn from a class."

Translation/ Clean Up

Select student full name
from ~~the~~ students ~~table~~ left outer join~~ed with~~
(Select student ID from ~~the~~ student schedules ~~table~~
inner join~~ed with the~~ student class status ~~table~~
on student_class_status.class status
~~in the student class status table matches~~
= student_schedules.class status ~~in the student schedules table~~
where class status description ~~is~~ = 'withdrew') as withdrew
on students.student ID ~~in the students table matches~~
= withdrew.student ID ~~in the embedded SELECT statement~~
where ~~the~~ student_schedules.student ID ~~in the student schedules table~~ is null

SQL

```
SELECT Students.StudLastName || ', ' ||
   Students.StudFirstName AS StudFullName
FROM Students
LEFT OUTER JOIN
   (SELECT Student_Schedules.StudentID
    FROM Student_Class_Status
    INNER JOIN Student_Schedules
    ON Student_Class_Status.ClassStatus =
       Student_Schedules.ClassStatus
    WHERE Student_Class_Status.ClassStatus
       Description = 'withdrew')
  AS Withdrew
ON Students.StudentID = Withdrew.StudentID
WHERE Withdrew.StudentID IS NULL
```

CH09_Students_Never_Withdrawn
(15 rows)

StudFullName
Hamilton, David
Stadick, Betsy
Galvin, Janice
Hartwig, Doris
Bishop, Scott
Hallmark, Elizabeth
Sheskey, Sara
Wier, Marianne
<< more rows here >>

"Show me all subject categories and any classes for all subjects."

Translation/ Clean Up

Select category description, subject name, classroom ID, start time, ~~and~~ duration
from ~~the~~ categories ~~table~~
left outer join~~ed with the~~ subjects ~~table~~
on categories.category ID ~~in the categories table matches~~
= subjects.category ID ~~in the subjects table,~~
~~then~~ left outer join~~ed with the~~ classes ~~table~~
on subjects .subject ID ~~in the subjects table matches~~
= classes.subject ID ~~in the classes table~~

SQL

```
SELECT Categories.CategoryDescription,
   Subjects.SubjectName, Classes.ClassroomID,
   Classes.StartTime, Classes.Duration
FROM (Categories
LEFT OUTER JOIN Subjects
ON Categories.CategoryID = Subjects.CategoryID)
LEFT OUTER JOIN Classes
ON Subjects.SubjectID = Classes.SubjectID
```

❖ **Note** We were very careful again to construct the sequence and nesting of JOINs to be sure we got the answer we expected.

CH09_All_Categories_All_Subjects_Any_Classes (82 rows)

CategoryDescription	SubjectName	ClassroomID	StartTime	Duration
Accounting	Financial Accounting Fundamentals I	3313	9:00	50
Accounting	Financial Accounting Fundamentals I	3313	13:00	50
Accounting	Financial Accounting Fundamentals II	3415	8:00	50
Accounting	Fundamentals of Managerial Accounting	3415	10:00	50
Accounting	Intermediate Accounting	3315	11:00	50
Accounting	Business Tax Accounting	3313	14:00	50
Art	Introduction to Art	1231	10:00	50
Art	Design	1619	15:30	110
<< more rows here >>				

Further down in the result set, you'll find no classes scheduled for Developing a Feasibility Plan, Computer Programming, and American Government. You'll also find no subjects scheduled for categories Psychology, French, or German.

Bowling League Database

"Show me tournaments that haven't been played yet."

Translation/ Clean Up
Select tourney ID, tourney date, ~~and~~ tourney location
from ~~the~~ tournaments ~~table~~
left outer join~~ed with the~~ tourney matches ~~table~~
on tournaments.tourney ID ~~in the tournaments table matches~~
= tourney_matches.tourney ID ~~in the tourney matches table~~
where match ID is null

SQL

```
SELECT Tournaments.TourneyID,
   Tournaments.TourneyDate,
   Tournaments.TourneyLocation
FROM Tournaments
LEFT OUTER JOIN Tourney_Matches
ON Tournaments.TourneyID =
   Tourney_Matches.TourneyID
WHERE Tourney_Matches.MatchID IS NULL
```

CH09_Tourney_Not_Yet_Played (6 rows)

TourneyID	TourneyDate	TourneyLocation
15	2008-07-11	Red Rooster Lanes
16	2008-07-18	Thunderbird Lanes
17	2008-07-25	Bolero Lanes
18	2008-08-01	Sports World Lanes
19	2008-08-08	Imperial Lanes
20	2008-08-15	Totem Lanes

"List all bowlers and any games they bowled over 180."

Translation 1
Select bowler name, tourney date, tourney location, match ID, and raw score from the bowlers table left outer joined with the bowler scores table on bowler ID, then inner joined with the tourney matches table on match ID, then finally inner joined with the tournaments table on tournament ID where raw score in the bowler scores table is greater than 180

Can you see why the above translation won't work? You need a filter on one of the tables that is on the right side of the left join, so you need to put the filter in an embedded SELECT statement. Let's restate the Translation step, clean it up, and solve the problem.

Translation 2/ Clean Up

Select bowler name, tourney date, tourney location,
match ID, ~~and~~ raw score
from ~~the~~ bowlers ~~table~~ left outer join~~ed with~~
(Select tourney date, tourney location, match ID,
bowler ID, ~~and~~ raw score
from ~~the~~ bowler scores ~~table~~
inner join~~ed with the~~ tourney matches ~~table~~
on bowler_scores .match ID ~~in the bowler scores table matches~~
= tourney_ matches.match ID ~~in the tourney matches table,~~
~~then~~ inner join~~ed with the~~ tournaments ~~table~~
on tournaments.tournament ID ~~in the tournaments table matches~~
= tourney_ matches.tournament ID ~~in the tourney matches table~~
where raw score ~~is greater than~~ > 180) as ti
on bowlers.bowler ID ~~in the bowlers table matches~~
= ti.bowler ID ~~in the embedded SELECT statement~~

SQL

```
SELECT Bowlers.BowlerLastName || ', ' ||
   Bowlers.BowlerFirstName AS BowlerName,
   TI.TourneyDate, TI.TourneyLocation,
   TI.MatchID, TI.RawScore
FROM Bowlers
LEFT OUTER JOIN
   (SELECT Tournaments.TourneyDate,
       Tournaments.TourneyLocation,
       Bowler_Scores.MatchID,
       Bowler_Scores.BowlerID,
       Bowler_Scores.RawScore
    FROM (Bowler_Scores
    INNER JOIN Tourney_Matches
    ON Bowler_Scores.MatchID =
       Tourney_Matches.MatchID)
    INNER JOIN Tournaments
    ON Tournaments.TourneyID =
       Tourney_Matches.TourneyID
    WHERE Bowler_Scores.RawScore > 180)
  AS TI
ON Bowlers.BowlerID = TI.BowlerID
```

CH09_All_Bowlers_And_Scores_Over_180 (106 rows)

BowlerName	TourneyDate	TourneyLocation	MatchID	RawScore
Black, Alastair				
Cunningham, David				
Ehrlich, Zachary				
Fournier, Barbara				
Fournier, David				
Hallmark, Alaina				
Hallmark, Bailey				
Hallmark, Elizabeth				
Hallmark, Gary				
Hernandez, Kendra				
Hernandez, Michael				
Kennedy, Angel	2007-11-20	Sports World Lanes	46	185
Kennedy, Angel	2007-10-09	Totem Lanes	22	182
<< more rows here >>				

❖ **Note** You guessed it! This is another example where you must build the filtered INNER JOIN result set first and then OUTER JOIN that with the table from which you want "all" rows.

Recipes Database

"List ingredients not used in any recipe yet."

Translation/ Clean Up

Select ingredient name from ~~the~~ ingredients ~~table~~
left outer join~~ed with the~~ recipe ingredients ~~table~~
on ingredients.ingredient ID ~~in the ingredients table matches~~
= recipe_ingredients.ingredient ID ~~in the recipe ingredients table~~
where recipe ID is null

SQL

```
SELECT Ingredients.IngredientName
FROM Ingredients
LEFT OUTER JOIN Recipe_Ingredients
ON Ingredients.IngredientID =
   Recipe_Ingredients.IngredientID
WHERE Recipe_Ingredients.RecipeID IS NULL
```

CH09_Ingredients_Not_Used (20 rows)

IngredientName
Halibut
Chicken, Fryer
Bacon
Iceberg Lettuce
Butterhead Lettuce
Scallop
Vinegar
Red Wine
<< *more rows here* >>

"I need all the recipe types, and then all the recipe names, and then any matching ingredient step numbers, ingredient quantities, and ingredient measurements, and finally all ingredient names from my recipes database."

Translation/ Clean Up

Select ~~the~~ recipe class description, recipe title,
ingredient name, recipe sequence number,
amount, ~~and~~ measurement description
from ~~the~~ recipe classes ~~table~~
full outer join~~ed with the~~ recipes ~~table~~
on recipe_classes.recipe class ID ~~in the recipe classes table matches~~
= recipes.recipe class ID ~~in the recipes table,~~
~~then~~ left outer join~~ed with the~~ recipe ingredients ~~table~~
on recipes.recipe ID ~~in the recipes table matches~~
= recipe_ingredients.recipe ID ~~in the recipe ingredients table,~~
~~then~~ inner join~~ed with the~~ measurements ~~table~~
on measurements.measurement amount ID
~~in the measurements table matches~~
= recipe_ ingredients.measurement amount ID
~~in the recipe ingredients table,~~
~~and then finally~~ full outer join~~ed with the~~ ingredients ~~table~~
on ingredients.ingredient ID ~~in the ingredients table matches~~
= recipe_ ingredients.ingredient ID ~~in the recipe ingredients table,~~

SQL

```
SELECT Recipe_Classes.RecipeClassDescription,
   Recipes.RecipeTitle,
   Ingredients.IngredientName,
   Recipe_Ingredients.RecipeSeqNo,
   Recipe_Ingredients.Amount,
   Measurements.MeasurementDescription
FROM (((Recipe_Classes
FULL OUTER JOIN Recipes
  ON Recipe_Classes.RecipeClassID =
     Recipes.RecipeClassID)
  LEFT OUTER JOIN Recipe_Ingredients
  ON Recipes.RecipeID =
     Recipe_Ingredients.RecipeID)
  INNER JOIN Measurements
  ON Measurements.MeasureAmountID =
     Recipe_Ingredients.MeasureAmountID)
  FULL OUTER JOIN Ingredients
  ON Ingredients.IngredientID =
     Recipe_Ingredients.IngredientID
ON Recipe_Classes.RecipeClassID =
   Recipes.RecipeClassID
```

❖ **Note** This sample is a request you saw us solve in the section on FULL OUTER JOIN. We decided to include it here so that you can see the actual result. You won't find this query saved using this syntax in the Microsoft Access or MySQL version of the sample database because neither product supports a FULL OUTER JOIN. Instead, you can find this problem solved with a UNION of two OUTER JOIN queries that achieves the same result. You'll learn about using UNION in the next chapter. The result shown here is what you'll see when you run the query in Microsoft SQL Server.

CH09_All_Recipe_Classes_All_Recipes (109 rows)

RecipeClass Description	RecipeTitle	Ingredient Name	RecipeSeq No	Amount	Measurement Description
Starch	Yorkshire Pudding	Flour	1	1.5	Cup
Starch	Yorkshire Pudding	Water	2	1	Cup
Starch	Yorkshire Pudding	Eggs	3	2	Piece
Starch	Yorkshire Pudding	Salt	4	0.5	Teaspoon
Starch	Yorkshire Pudding	Milk	5	0.5	Cup
Starch	Yorkshire Pudding	Beef drippings	6	4	Teaspoon
Dessert	Trifle	Sponge Cake	1	1	Package
Dessert	Trifle	Raspberry Jello	2	1	Package
Dessert	Trifle	Bird's Custard Powder	3	1	Package
Dessert	Trifle	Raspberry Jam	4	1	Jar
<< more rows here >>					

SUMMARY

In this chapter, we led you through the world of OUTER JOINs. We began by defining an OUTER JOIN and comparing it to the INNER JOIN you learned about in Chapter 8.

We next explained how to construct a LEFT or RIGHT OUTER JOIN, beginning with simple examples using two tables, and then progressing to embedding SELECT statements and constructing statements using multiple JOINs. We showed how an OUTER JOIN combined with a Null test is equivalent to the difference (EXCEPT) operation we covered in Chapter 7. We also discussed some of the difficulties you might encounter when constructing statements using multiple OUTER JOINs. We closed the discussion of the LEFT and RIGHT OUTER JOIN with a problem requiring multiple OUTER JOINs that can't be solved with only LEFT or RIGHT.

In our discussion of FULL OUTER JOIN, we showed how you might need to use this type of JOIN in combination with other INNER and OUTER JOINs to get the correct answer. We also briefly explained a variant of the FULL OUTER JOIN—the UNION JOIN.

We explained how OUTER JOINs are useful and listed a variety of requests that you can solve using OUTER JOINs. The rest of the chapter showed nearly a dozen examples of how to use OUTER JOIN. We provided several examples for each of the sample databases and showed you the logic behind constructing the solution statement for each request.

The following section presents a number of requests that you can work out on your own.

Problems for You to Solve

Below, we show you the request statement and the name of the solution query in the sample databases. If you want some practice, you can work out the SQL you need for each request and then check your answer with the query we saved in the samples. Don't worry if your syntax doesn't exactly match the syntax of the queries we saved—as long as your result set is the same.

Sales Orders Database

1. *"Show me customers who have never ordered a helmet."*
   (Hint: This is another request where you must first build an INNER JOIN to find all orders containing helmets and then do an OUTER JOIN with Customers.)
   You can find the solution in CH09_Customers_No_Helmets (2 rows).
2. *"Display customers who have no sales rep (employees) in the same ZIP Code."*
   You can find the solution in CH09_Customers_No_Rep_Same_Zip (18 rows).
3. *"List all products and the dates for any orders."*
   You can find the solution in CH09_All_Products_Any_Order_Dates (2,682 rows).

Entertainment Agency Database

1. *"Display agents who haven't booked an entertainer."*
   You can find the solution in Agents_No_Contracts (1 row).
2. *"List customers with no bookings."*
   You can find the solution in CH09_Customers_No_Bookings (2 rows).
3. *"List all entertainers and any engagements they have booked."*
   You can find the solution in CH09_All_Entertainers_And_Any_Engagements (112 rows).

School Scheduling Database

1. *"Show me classes that have no students enrolled."*
   (Hint: You need only "enrolled" rows from Student_Classes, not "completed" or "withdrew.")
   You can find the solution in CH09_Classes_No_Students_Enrolled (63 rows).
2. *"Display subjects with no faculty assigned."*
   You can find the solution in CH09_Subjects_No_Faculty (1 row).
3. *"List students not currently enrolled in any classes."*
   (Hint: You need to find which students have an "enrolled" class status in student schedules and then find the students who are not in this set.)
   You can find the solution in CH09_Students_Not_Currently_Enrolled (2 rows).
4. *"Display all faculty and the classes they are scheduled to teach."*
   You can find the solution in CH09_All_Faculty_And_Any_Classes (79 rows).

Bowling League Database

1. *"Display matches with no game data."*
   You can find the solution in CH09_Matches_Not_Played_Yet (1 row).
2. *"Display all tournaments and any matches that have been played."*
   You can find the solution in CH09_All_Tourneys_Match_Results (174 rows).

Recipes Database

1. *"Display missing types of recipes."*
   You can find the solution in CH09_Recipe_Classes_No_Recipes (1 row).
2. *"Show me all ingredients and any recipes they're used in."*
   You can find the solution in CH09_All_Ingredients_Any_Recipes (108 rows).
3. *"List the salad, soup, and main course categories and any recipes."*
   You can find the solution in CH09_Salad_Soup_Main_Courses (9 rows).
4. *"Display all recipe classes and any recipes."*
   You can find the solution in CH09_All_RecipesClasses_And_Matching_Recipes (16 rows).

This page intentionally left blank

10

UNIONs

"I beseech those whose piety will permit them reverently to petition, that they will pray for this union."
—Sam Houston, *Texas hero*

Topics Covered in This Chapter

What Is a UNION?

Writing Requests with UNION

Uses for UNION

Sample Statements

Summary

Problems for You to Solve

In Chapter 7, Thinking in Sets, we introduced three fundamental set operations—intersection, difference, and union. Chapter 8, INNER JOINs, showed how to perform the equivalent of an intersection operation by linking result sets on key values using INNER JOIN. Chapter 9, OUTER JOINs, discussed how to ask for a set difference by using an OUTER JOIN and testing for the Null value. This chapter explains how to do the third operation, a UNION.

What Is a UNION?

A UNION lets you select the rows from two or more similar result sets and combine them into a single result set. Notice that we said "rows," not "columns." In Chapters 8 and 9, you learned how to bring together columns from two or more result sets using a JOIN. When you ask for a JOIN, the columns from the result sets appear side by side. For example, if you ask for the RecipeClassDescription from the Recipe_Classes table and the

RecipeTitle from the Recipes table with a JOIN, you get a result set that looks like Figure 10–1.

RecipeClassDescription	RecipeTitle
Main course	Irish Stew
Main course	Fettuccini Alfredo
Main course	Pollo Picoso
Main course	Roast Beef
Main course	Huachinango Veracruzana (Red Snapper, Veracruz style)
Main course	Tourtière (French-Canadian Pork Pie)
Main course	Salmon Filets in Parchment Paper
Vegetable	Garlic Green Beans
<< more rows here >>	

Figure 10–1 *Fetching data from two tables using a JOIN*

Let's first take a quick look at the syntax for a basic UNION, as shown in Figure 10–2.

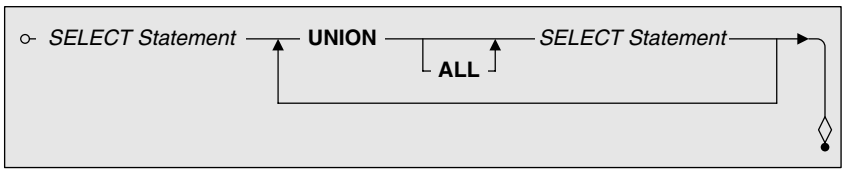


Figure 10–2 *The syntax diagram for a basic UNION statement*

A UNION interleaves the rows from one result set with the rows from another result set. You define each result set by writing a SELECT statement that can include not only a complex JOIN in the FROM clause but also WHERE, HAVING, and GROUP BY clauses. You then link them with the UNION keyword. (You'll learn about the GROUP BY clause in Chapter 13, Grouping Data, and the HAVING clause in Chapter 14, Filtering Grouped Data.) If you ask for RecipeClassDescription from the Recipe_Classes table UNION RecipeTitle from the Recipes table, you get an answer that looks like Figure 10–3.

Notice that we get only one column in the result set. The name of the column is inherited from the column in the first table we chose to include in the SELECT expression, but it includes information on both RecipeTitle (Asparagus) and RecipeClassDescription (Dessert). Instead of appearing side by side, the data from the two columns is interleaved vertically.

RecipeClassDescription
Asparagus
Coupe Colonel
Dessert
Fettuccini Alfredo
Garlic Green Beans
Hors d'oeuvres
Huachinango Veracruzana (Red Snapper, Veracruz style)
Irish Stew
<< more rows here >>

Figure 10–3 *Fetching data from two tables using a UNION*

If you studied the diagram in Figure 10–2, you're probably wondering what the optional keyword ALL is about. When you leave out that keyword, your database system eliminates any rows that have duplicate values. For example, if there's a RecipeClassDescription of Dessert and a RecipeTitle of Dessert, you get only one Dessert row in the final result set. Conversely, when you include the ALL keyword, no duplicate rows are removed. Note that UNION ALL is likely to be much more efficient because your database system doesn't have to do extra work to look for and eliminate any duplicate rows. If you're certain that the queries you are combining with UNION don't contain any duplicate rows (or you don't care about duplicates), then always use the ALL keyword.

To perform a UNION, the two result sets must meet certain requirements. First, each of the two SELECT statements that you're linking with a UNION must have the same number of output columns specified after the SELECT keyword so that the result set will have the same number of columns. Secondly, each corresponding column must be what the SQL Standard calls "comparable."

❖ **Note** The full SQL:2003 Standard allows you to UNION dissimilar sets. However, most commercial implementations support the basic or entry-level standard we're describing here. You might find that your database system allows you to use UNION in more creative ways.

As discussed in Chapter 6, Filtering Your Data, you should compare only character values with character values, number values with number values, or datetime values with datetime values. Although some database systems allow

mixing data types in a comparison, it really doesn't make sense to compare a character value such as "John" to a numeric value such as 55. If it makes sense to compare two columns in a WHERE clause, then the columns are comparable. This is what the SQL Standard means when it requires that a column from one result set that you want to UNION with a column from another result set must be of a comparable data type.

Writing Requests with UNION

In the previous chapters on INNER JOIN and OUTER JOIN, we studied how to construct a SELECT statement using the SELECT, FROM, and WHERE clauses. The focus of those two chapters was on constructing complex JOINs within the FROM clause. To construct a UNION, you now have to graduate to a *SELECT expression* that links two or more SELECT statements with the UNION operator. Each SELECT statement can have as simple or complex a FROM clause as you need to get the job done.

Using Simple SELECT Statements

Let's start simply by creating a UNION of two simple SELECT statements that use a single table in the FROM clause. Figure 10–4 shows the syntax diagram for a UNION of two simple SELECT statements.

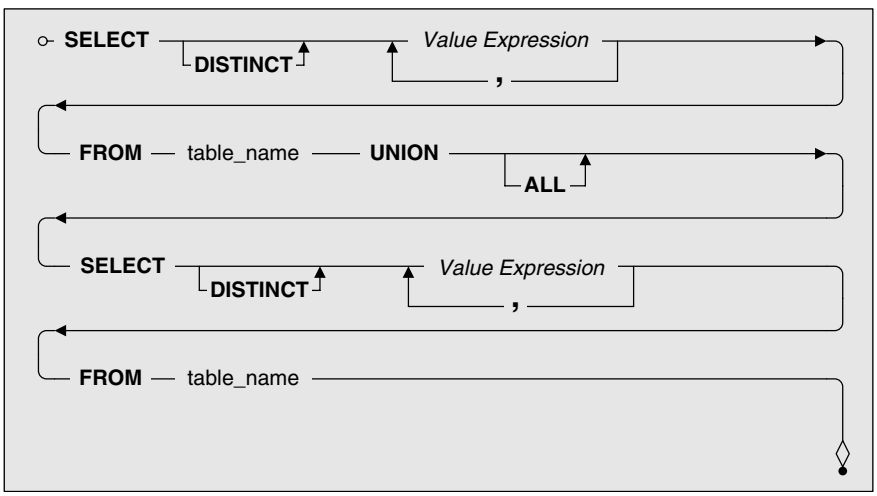


Figure 10–4 *Using a UNION to combine two simple SELECT statements*

Unlike when you ask for a JOIN, all the action happens in the UNION operator that you specify to combine the two SELECT statements. As mentioned earlier, if you leave out the optional ALL keyword, your database system eliminates any duplicate rows it finds. This means that the result set from your request might have fewer rows than the sum of the number of rows returned from each result set participating in the UNION. On the other hand, if you include the ALL keyword, the number of rows in the result set will be equal to the sum of the number of rows in the two participating result sets.

❖ **Note** The SQL Standard also defines a CORRESPONDING clause that you can place after the UNION keyword to indicate that you want the UNION performed by comparing columns that have the same name in each result set. You can also further restrict the comparison set by including a specific list of column names after the CORRESPONDING keyword. We could not find a major commercial implementation of this feature, but you might find it supported in future releases of the product you use.

Let's create a simple UNION—a mailing list for customers and vendors from the Sales Orders sample database. Figure 10–5 shows the two tables needed.

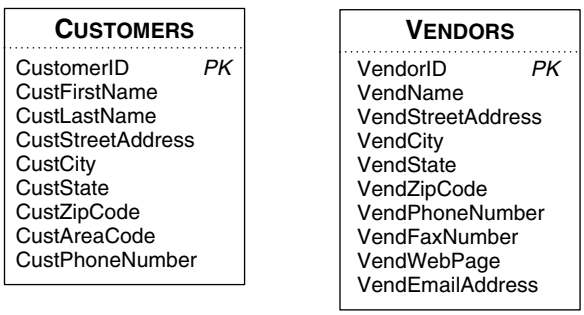


Figure 10–5 *The Customers and Vendors tables from the Sales Orders sample database*

Notice that there's no "natural" relationship between these two tables, but they do both contain columns that have similar meanings and data types. In a mailing list, you need a name, street address, city, state, and ZIP Code. Because all these fields in both tables are comparable character data, we don't need to worry about data types. (Some database designers might make ZIP Code a number, but that's OK too, as long as the ZIP Code column from one table is

a data type that's comparable with the data type of the ZIP Code column from the second table.)

One problem is that the name in the Vendors table is a single column, but there are two name fields in Customers: CustFirstName and CustLastName. In order to come up with the same number of columns from both tables, we need to build an expression on the two columns from Customers to create a single column expression to UNION with the single name column from Vendors. Let's build the query.

❖ **Note** Throughout this chapter, we use the "Request/Translation/Clean Up/SQL" technique introduced in Chapter 4, Creating a Simple Query.

"Build a single mailing list that consists of the name, address, city, state, and ZIP Code for customers and the name, address, city, state, and ZIP Code for vendors."

Translation: Select customer full name, customer address, customer city, customer state, and customer ZIP Code from the customers table combined with vendor name, vendor address, vendor city, vendor state, and vendor ZIP Code from the vendors table

Clean Up: Select customer full name, customer address, customer city, customer state, ~~and~~ customer ZIP Code
from ~~the~~ customers ~~table~~
~~combined with~~ union Select vendor name, vendor address, vendor city, vendor state, ~~and~~ vendor ZIP Code
from ~~the~~ vendors ~~table~~

SQL:

```
SELECT Customers.CustLastName || ', ' ||
   Customers.CustFirstName AS MailingName,
   Customers.CustStreetAddress, Customers.CustCity,
   Customers.CustState, Customers.CustZipCode
FROM Customers UNION SELECT Vendors.VendName,
   Vendors.VendStreetAddress, Vendors.VendCity,
   Vendors.VendState, Vendors.VendZipCode
FROM Vendors
```

Notice that each SELECT statement generates five columns, but we had to use an expression to combine the two name columns in the Customers table into a single column. All the columns from both SELECT statements are character data, so we have no problem with their being comparable.

You might be wondering: "What are the names of the columns that are output from this query?" Good question! The SQL Standard specifies that when the names of respective columns are the same (for example, the name of the fourth column of the first SELECT statement is the same as the name of the fourth column of the second SELECT statement), that's the name of the output column. If the column names are different (as in the example we just constructed), the SQL Standard states: "If a <query expression body> immediately contains UNION or INTERSECT, and the <column name>s of a pair of corresponding columns of the operand tables are not equivalent, then the result column has an implementation-dependent <column name>."

In plain English, this means that your database system decides what names to assign to the output columns. Your system is compliant with the SQL Standard as long as the name doesn't appear in some other column position in one of the result sets participating in the UNION. Most commercial database systems default to the names of the columns in the first SELECT statement. For the previous example, this means that you'll see column names of MailingName, CustStreetAddress, CustCity, CustState, and CustZipCode.

Notice that we did not include the ALL keyword in the UNION. Although it is unlikely that a customer last name and first name will match a vendor name (never mind the address, city, state, and ZIP Code), we wanted to avoid duplicate mailing addresses. If you're certain that you won't have any duplicates in two or more UNION sets, you can include the ALL keyword. Using ALL most likely will cause the request to run faster because your database system won't have to do extra work attempting to remove duplicates.

Combining Complex SELECT Statements

As you might imagine, the SELECT statements you combine with a UNION operator can be as complex as you need to get the job done. The only restriction is that both SELECT statements must ultimately provide the same number of columns, and the columns in each relative position must be comparable data types.

Suppose you want a list of all the customers and the bikes they ordered combined with all the vendors and the bikes they supply. First, let's identify all the tables we need. Figure 10–6 (on page 346) shows the tables needed to link customers to products.

Looks like we need to JOIN four tables. If we want to find vendors and the products they sell, we need the tables shown in Figure 10–7 (on page 346).

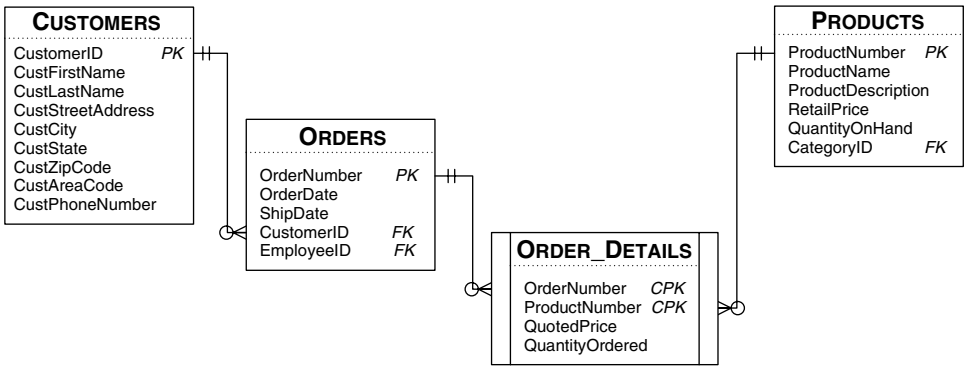


Figure 10–6 *Table relationships to link customers to the products they ordered*

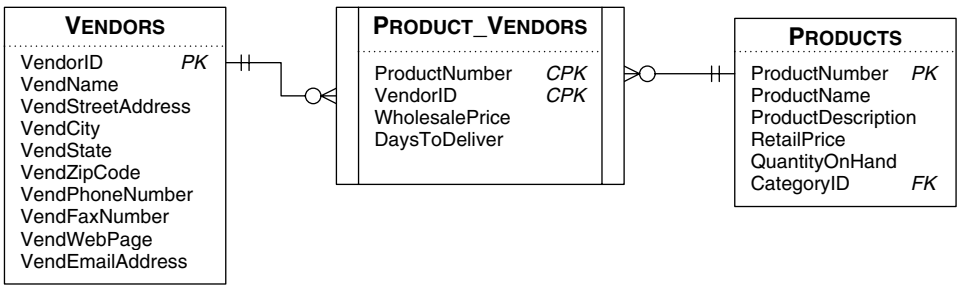


Figure 10–7 *Table relationships to link vendors to the products they sell*

As discussed in Chapter 8, you can nest multiple JOIN clauses to link several tables to gather the information you need to solve a complex problem. For review, Figure 10–8 (on page 348) shows the syntax for nesting three tables.

We now have all the pieces needed to solve the puzzle. We can build a compound INNER JOIN to fetch the customer information, insert a UNION keyword, and then build the compound INNER JOIN for the vendor information.

"List customers and the bikes they ordered combined with vendors and the bikes they sell."

Translation Select customer full name and product name from the customers table joined with the orders table on customer ID in the customers table matches customer ID in the orders table, then joined with the order details table on order number in the orders table matches order number in the order details table, and then joined with the products table on product number in

the products table matches product number in the order details table where product name contains 'bike', combined with select vendor name and product name from the vendors table joined with the product vendors table on vendor ID in the vendors table matches vendor ID in the product vendors table, and then joined with the products table on product number in the products table matches product number in the product vendors table where product name contains 'bike'

Clean Up

Select customer full name ~~and~~ product name
from ~~the~~ customers ~~table~~ join~~ed with the~~ orders ~~table~~
on customers.customer ID ~~in the customers table matches~~
= orders.customer ID ~~in the orders table,~~
~~then~~ join~~ed with the~~ order details ~~table~~
on orders.order number ~~in the orders table matches~~
= order_details.order number ~~in the order details table,~~
~~and then~~ join~~ed with the~~ products ~~table~~
on products.product number ~~in the products table matches~~
= order_details.product number ~~in the order details table~~
where product name ~~contains~~ like '%bike%'~~,~~
~~combined with~~ union select vendor name ~~and~~ product name
from ~~the~~ vendors ~~table~~ join~~ed with the~~ product vendors ~~table~~
on vendors.vendor ID ~~in the vendors table matches~~
= product_vendors.vendor ID ~~in the product vendors table,~~
~~and then~~ join~~ed with the~~ products ~~table~~
on products.product number ~~in the products table matches~~
= product_vendors.product number ~~in the product vendors table~~
where product name ~~contains~~ like '%bike%'

SQL

```
SELECT Customers.CustLastName || ', ' ||
   Customers.CustFirstName AS FullName,
   Products.ProductName, 'Customer' AS RowID
FROM ((Customers INNER JOIN Orders
ON Customers.CustomerID = Orders.CustomerID)
INNER JOIN Order_Details
ON Orders.OrderNumber = Order_Details.OrderNumber)
INNER JOIN Products
ON Products.ProductNumber =
   Order_Details.ProductNumber
WHERE Products.ProductName LIKE '%bike%'
UNION
SELECT Vendors.VendName, Products.ProductName,
   'Vendor' AS RowID
```

```
FROM (Vendors INNER JOIN Product_Vendors
ON Vendors.VendorID = Product_Vendors.VendorID)
INNER JOIN Products
ON Products.ProductNumber =
   Product_Vendors.ProductNumber
WHERE Products.ProductName LIKE '%bike%'
```

Well, that's about the size of the King Ranch, but it gets the job done! Notice that we also threw in a character string literal that we named RowID in both SELECT statements so that it will be easy to see which rows originate from customers and which ones come from vendors. You might be tempted to insert a DISTINCT keyword in the first SELECT statement because a really good customer might have ordered a particular bike model more than once. Because we didn't use the ALL keyword on the UNION, the request will eliminate any duplicates anyway. If you add DISTINCT, you might be asking your database system to perform extra work to eliminate duplicates twice!

When you need to build a UNION query, we recommend that you build the separate SELECT statements first. It's easy then to copy and paste the syntax for each SELECT statement into a new query, separating each statement with the UNION keyword.

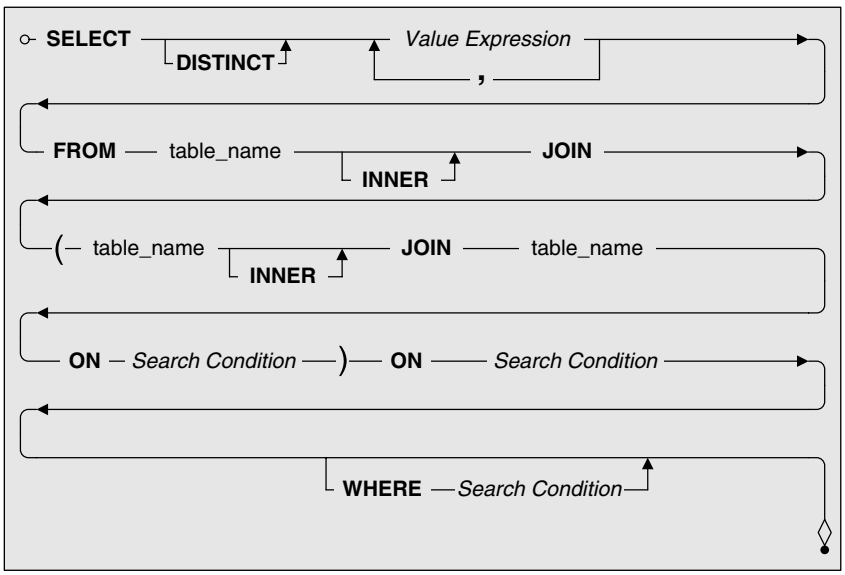


Figure 10–8 *The syntax for JOINing three tables*

Using UNION More Than Once

So far, we have shown you only how to use a UNION to combine two result sets. In truth, you can follow the second SELECT statement specification with another UNION keyword and another SELECT statement. Although some implementations limit the number of result sets you can combine with UNION, in theory you can keep adding UNION SELECT to your heart's content.

Suppose you need to build a single mailing list from three different tables—Customers, Employees, and Vendors—perhaps to create a combined list for holiday greeting labels. Figure 10–9 shows a diagram of the syntax to build this list.

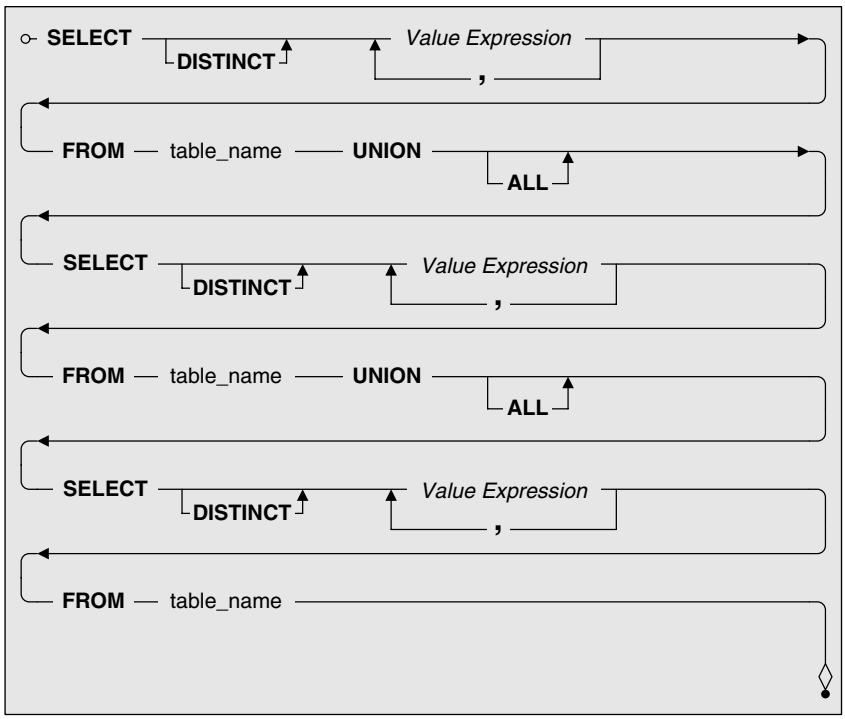


Figure 10–9 *Creating a UNION of three tables*

You can see that you need to create one SELECT statement to fetch all the names and addresses from the Customers table, UNION that with a SELECT statement for the same information from the Employees table, and finally UNION that with a SELECT statement for names and addresses from the

Vendors table. (To simplify the process, we have combined the Translation and Clean Up steps in this example.)

"Create a single mailing list for customers, employees, and vendors."

Translation/ Clean Up

Select customer full name, customer street address, customer city, customer state, ~~and~~ customer ZIP Code
from ~~the~~ customers ~~table~~
~~combined with~~ union
select employee full name, employee street address, employee city, employee state, ~~and~~ employee ZIP Code
from ~~the~~ employees ~~table~~
~~combined with~~ union
select vendor name, vendor street address, vendor city, vendor state, ~~and~~ vendor ZIP Code
from ~~the~~ vendors ~~table~~

SQL

```
SELECT Customers.CustFirstName || ' ' ||
   Customers.CustLastName AS CustFullName,
   Customers.CustStreetAddress, Customers.CustCity,
   Customers.CustState, Customers.CustZipCode
FROM Customers
UNION
SELECT Employees.EmpFirstName || ' ' ||
   Employees.EmpLastName AS EmpFullName,
   Employees.EmpStreetAddress, Employees.EmpCity,
   Employees.EmpState, Employees.EmpZipCode
FROM Employees
UNION
SELECT Vendors.VendName, Vendors.VendStreetAddress,
   Vendors.VendCity, Vendors.VendState,
   Vendors.VendZipCode
FROM Vendors
```

Of course, if you want to filter the mailing list for a particular city, state, or range of ZIP Codes, you can add a WHERE clause to any or all of the SELECT statements. If, for example, you want to create a list for the customers, employees, and vendors only in a particular state, you must add a WHERE clause to *each* of the embedded SELECT statements. You could also apply a filter to just one of the SELECT statements, for example, to create a list for vendors in the state of Texas combined with all customers and all employees.

Sorting a UNION

What about sorting the result of a UNION? You'll find on many database systems that the result set appears as though it is sorted by the output columns from left to right. For example, in the UNION of three tables we just built in the previous section, the rows will appear in sequence first by name, then by street address, and so on.

To keep the postal service happy (and perhaps get a discount for a large mailing), sort your rows by ZIP Code. You can add an ORDER BY clause to do this, but the trick is that this clause must appear at the very end after the last SELECT statement. The ORDER BY applies to the result of the UNION, not the last SELECT statement. Figure 10–10 shows how to do this.

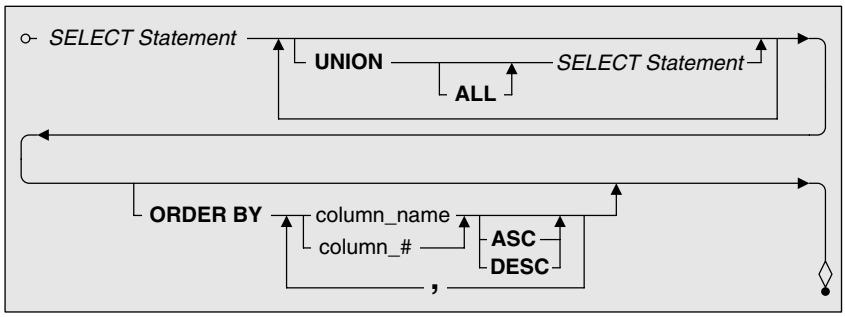


Figure 10–10 *Adding a sorting specification to a UNION query*

As the diagram shows, you can loop through a UNION SELECT statement as many times as you like to pick up all the result sets you need to combine, but the ORDER BY clause must appear at the end. You might ask, "What do I use for column_name or column_# in the ORDER BY clause?" Remember that you're sorting the output of all the previous parts of the SELECT expression. As discussed earlier, the output names of the columns are "implementation-dependent," but most database systems use the column names generated by the first SELECT statement.

You can also specify the relative column number, starting with 1, as the first output column. In a query that outputs name, street address, city, state, and ZIP Code, you need to specify a column_# of 5 (ZIP Code is the fifth column) to sort by zip.

Let's sort the mailing list query using both techniques. Here's the correct syntax for sorting by column name.

```
SQL         SELECT Customers.CustFirstName || ' ' ||
               Customers.CustLastName AS CustFullName,
               Customers.CustStreetAddress, Customers.CustCity,
               Customers.CustState, Customers.CustZipCode
            FROM Customers
            UNION
            SELECT Employees.EmpFirstName || ' ' ||
               Employees.EmpLastName AS EmpFullName,
               Employees.EmpStreetAddress, Employees.EmpCity,
               Employees.EmpState, Employees.EmpZipCode
            FROM Employees
            UNION
            SELECT Vendors.VendName, Vendors.VendStreetAddress,
               Vendors.VendCity, Vendors.VendState,
               Vendors.VendZipCode
            FROM Vendors
            ORDER BY CustZipCode
```

Of course, we're assuming that the name of the output column we want to sort is the name of the column from the first SELECT statement. Using a relative column number to specify the sort looks like this.

```
SQL         SELECT Customers.CustFirstName || ' ' ||
               Customers.CustLastName AS CustFullName,
               Customers.CustStreetAddress, Customers.CustCity,
               Customers.CustState, Customers.CustZipCode
            FROM Customers
            UNION
            SELECT Employees.EmpFirstName || ' ' ||
               Employees.EmpLastName AS EmpFullName,
               Employees.EmpStreetAddress, Employees.EmpCity,
               Employees.EmpState, Employees.EmpZipCode
            FROM Employees
            UNION
            SELECT Vendors.VendName, Vendors.VendStreetAddress,
               Vendors.VendCity, Vendors.VendState,
               Vendors.VendZipCode
            FROM Vendors
            ORDER BY 5
```

Uses for UNION

You probably won't use UNION as much as INNER JOIN and OUTER JOIN. You most likely will use UNION to combine two or more similar result sets from different tables. Although you *can* use UNION to combine two result sets from the same table or set of tables, you usually can solve those sorts of

problems with a simple SELECT statement containing a more complex WHERE clause. We include a couple of examples in the Sample Statements section and show you the more efficient way to solve the same problem with a WHERE clause instead of a UNION.

Here's just a small sample of the types of problems you can solve with UNION using the sample databases.

"Show me all the customer and employee names and addresses."

"List all the customers who ordered a bicycle combined with all the customers who ordered a helmet." (This is one of those problems that can also be solved with a single SELECT statement and a complex WHERE clause.)

"Produce a mailing list for customers and vendors."

"List the customers who ordered a bicycle together with the vendors who provide bicycles."

"Create a list that combines agents and entertainers."

"Display a combined list of customers and entertainers."

"Produce a list of customers who like contemporary music together with a list of entertainers who play contemporary music."

"Create a mailing list for students and staff."

"Show me the students who have an average score of 85 or better in Art together with the faculty members who teach Art and have a proficiency rating of 9 or better."

"Find the bowlers who had a raw score of 155 or better at Thunderbird Lanes combined with bowlers who had a raw score of 140 or better at Bolero Lanes." (This is another problem that can also be solved with a single SELECT statement and a complex WHERE clause.)

"List the tourney matches, team names, and team captains for the teams starting on the odd lane together with the tourney matches, team names, and team captains for the teams starting on the even lane."

"Create an index list of all the recipe titles and ingredients."

"Display a list of all ingredients and their default measurement amounts together with ingredients used in recipes and the measurement amount for each recipe."

Sample Statements

You now know the mechanics of constructing queries using UNION and have seen some of the types of requests you can answer with a UNION. Let's take a look at a fairly robust set of samples using UNION from each of the sample

databases. These examples illustrate the use of the UNION operation to combine sets of rows.

We've also included sample result sets that would be returned by these operations and placed them immediately after the SQL syntax line. The name that appears immediately above a result set is the name we gave each query in the sample data on the companion CD you'll find bound into the back of the book. We stored each query in the appropriate sample database (as indicated within the example), and we prefixed the names of the queries relevant to this chapter with "CH10." You can follow the instructions in the Introduction of this book to load the samples onto your computer and try them.

❖ **Note** Because many of these examples use complex joins, the optimizer for your database system might choose a different way to solve these queries. For this reason, the first few rows might not exactly match the result you obtain, but the total number of rows should be the same. To simplify the process, we have combined the Translation and Clean Up steps for all the following examples.

Sales Orders Database

"Show me all the customer and employee names and addresses, including any duplicates, sorted by ZIP Code."

Translation/ Clean Up

```
Select customer first name, customer last name,
customer street address, customer city,
customer state, and customer ZIP Code
from the customers table
combined with union all
Select employee first name, employee last name,
employee street address, employee city,
employee state, and employee ZIP Code
from the employees table,
order by ZIP Code
```

```
SQL         SELECT Customers.CustFirstName,
               Customers.CustLastName,
               Customers.CustStreetAddress, Customers.CustCity,
               Customers.CustState, Customers.CustZipCode
            FROM Customers
            UNION ALL
            SELECT Employees.EmpFirstName,
               Employees.EmpLastName,
               Employees.EmpStreetAddress, Employees.EmpCity,
               Employees.EmpState, Employees.EmpZipCode
            FROM Employees
            ORDER BY CustZipCode
```

CH10_Customers_UNION_ALL_Employees (35 rows)

CustFirst Name	CustLast Name	CustStreet Address	CustCity	CustState	CustZip Code
Estella	Pundt	2500 Rosales Lane	Dallas	TX	75260
Robert	Brown	672 Lamont Ave	Houston	TX	77201
Kirk	DeGrasse	455 West Palm Ave	San Antonio	TX	78284
Kirk	DeGrasse	455 West Palm Ave	San Antonio	TX	78284
Angel	Kennedy	667 Red River Road	Austin	TX	78710
Maria	Patterson	3445 Cheyenne Road	El Paso	TX	79915
Mark	Rosales	323 Advocate Lane	El Paso	TX	79915
Caleb	Viescas	4501 Wetland Road	Long Beach	CA	90809
<< more rows here >>					

(Notice that Kirk DeGrasse must be both a customer and an employee.)

"List all the customers who ordered a bicycle combined with all the customers who ordered a helmet."

Translation/ Clean Up

Select customer first name, customer last name,
~~and the constant~~ 'Bike'
from ~~the~~ customers ~~table~~
join~~ed with the~~ orders ~~table~~
on customers.customer ID ~~in the customers table matches~~
= orders.customer ID ~~in the orders table,~~
~~then~~ join~~ed with the~~ order details ~~table~~
on orders.order number ~~in the orders table matches~~
= order_details.order number ~~in the order details table,~~
~~and then~~ join~~ed with the~~ products ~~table~~
on product number ~~in the products table matches~~
= order_details.product number ~~in the order details table~~
where product name ~~contains~~ like '%bike%,'
~~combined with~~ union
Select customer first name, customer last name,
~~and the~~ ~~constant~~ 'Helmet'
from ~~the~~ customers ~~table~~ join~~ed with the~~ orders ~~table~~
on customers.customer ID ~~in the customers table matches~~
= orders.customer ID ~~in the orders table,~~
~~then~~ join~~ed with the~~ order details ~~table~~
on orders.order number ~~in the orders table matches~~
= order_details.order number ~~in the order details table,~~
~~and then~~ join~~ed with the~~ products ~~table~~
on product number ~~in the products table matches~~
= order_details.product number ~~in the order details table~~
where product name ~~contains~~ like '%helmet%'

SQL

```
SELECT Customers.CustFirstName,
   Customers.CustLastName, 'Bike' AS ProdType
FROM ((Customers
INNER JOIN Orders
ON Customers.CustomerID = Orders.CustomerID)
INNER JOIN Order_Details
ON Orders.OrderNumber = Order_Details.OrderNumber)
INNER JOIN Products
ON Products.ProductNumber =
   Order_Details.ProductNumber
WHERE Products.ProductName LIKE '%bike%'
UNION
SELECT Customers.CustFirstName,
```

```
   Customers.CustLastName, 'Helmet' AS ProdType
FROM ((Customers INNER JOIN Orders
ON Customers.CustomerID = Orders.CustomerID)
INNER JOIN Order_Details
ON Orders.OrderNumber = Order_Details.OrderNumber)
INNER JOIN Products
ON Products.ProductNumber =
   Order_Details.ProductNumber
WHERE Products.ProductName LIKE '%helmet%'
```

CH10_Customer_Order_Bikes_UNION_ Customer_Order_Helmets (52 rows)

CustFirstName	CustLastName	ProdType
Alaina	Hallmark	Bike
Andrew	Cencini	Bike
Andrew	Cencini	Helmet
Angel	Kennedy	Bike
Angel	Kennedy	Helmet
Caleb	Viescas	Bike
Caleb	Viescas	Helmet
Darren	Gehring	Bike
<< *more rows here* >>		

Notice that this is one of those problems that can also be solved with a single SELECT statement and a slightly more complex WHERE clause. The one advantage of using a UNION is that it's easy to add an artificial "set identifier" column (in this case, the ProdType column) to each result set so that you can see which customers came from which result set. However, most database systems solve a WHERE clause—even one with complex criteria—much faster than they solve a UNION. Following is the SQL to solve the same problem with a WHERE clause.

```
SQL         SELECT DISTINCT Customers.CustFirstName,
               Customers.CustLastName
            FROM
               ((Customers INNER JOIN Orders
            ON Customers.CustomerID = Orders.CustomerID)
            INNER JOIN Order_Details
            ON Orders.OrderNumber = Order_Details.OrderNumber)
            INNER JOIN Products
            ON Products.ProductNumber =
               Order_Details.ProductNumber
            WHERE Products.ProductName LIKE '%bike%'
            OR Products.ProductName LIKE '%helmet%'
```

CH10_Customers_Bikes_Or_Helmets (27 rows)

CustFirstName	CustLastName
Alaina	Hallmark
Andrew	Cencini
Angel	Kennedy
Caleb	Viescas
Darren	Gehring
David	Smith
Dean	McCrae
Estella	Pundt
<< more rows here >>	

❖ **Note** You can see that you need a DISTINCT keyword to eliminate duplicates when you don't use UNION. Remember that UNION automatically eliminates duplicates unless you specify UNION ALL. You can specify DISTINCT in the UNION examples, but you're asking your database system to do more work than necessary.

Entertainment Agency Database

"Create a list that combines agents and entertainers."

Translation/ Clean Up

Select agent full name, ~~and the constant~~ 'Agent'
from ~~the~~ agents ~~table~~
~~combined with~~ union
Select entertainer stage name, ~~and the constant~~ 'Entertainer'
from ~~the~~ entertainers ~~table~~

SQL

```
SELECT Agents.AgtLastName || ', ' ||
   Agents.AgtFirstName AS Name, 'Agent' AS Type
FROM Agents
UNION
SELECT Entertainers.EntStageName,
   'Entertainer' AS Type
FROM Entertainers
```

CH10_Agents_UNION_Entertainers (22 rows)

Name	Type
Bishop, Scott	Agent
Carol Peacock Trio	Entertainer
Caroline Coie Cuartet	Entertainer
Coldwater Cattle Company	Entertainer
Country Feeling	Entertainer
Dumbwit, Daffy	Agent
Jazz Persuasion	Entertainer
Jim Glynn	Entertainer
<< more rows here >>	

School Scheduling Database

"Show me the students who have a grade of 85 or better in Art together with the faculty members who teach Art and have a proficiency rating of 9 or better."

Translation/ Clean Up

Select student first name ~~aliased~~ as FirstName, student last name ~~aliased~~ as LastName, ~~and~~ grade ~~aliased~~ as Score
from ~~the~~ students ~~table~~
join~~ed with the~~ student schedules ~~table~~
on students.student ID ~~in the students table matches~~
= student_schedules.student ID ~~in the student schedules table, then~~ join~~ed with the~~ student class status ~~table~~
on student_class_status.class status
~~in the student class status table matches~~
= student_schedules.class status ~~in the student schedules table, then~~ join~~ed with the~~ classes ~~table~~
on classes.class ID ~~in the classes table matches~~
= student_schedules.class ID ~~in the student schedules table, and then~~ join~~ed with the~~ subjects ~~table~~
on subjects.subject ID ~~in the subjects table matches~~
= classes.subject ID ~~in the classes table~~
where class status description ~~is~~ = 'completed'
and grade ~~is greater than or equal to~~ >= 85
and category ID ~~is~~ = 'ART' ~~combined with~~
union Select staff first name, staff last name,
~~and~~ proficiency rating ~~aliased~~ as Score
from ~~the~~ staff ~~table~~ join~~ed with the~~ faculty subjects ~~table~~
on staff.staff ID ~~in the staff table matches~~
= faculty_subjects.staff ID ~~in the faculty subjects table, and then~~ join~~ed with the~~ subjects ~~table~~
on subjects.subject ID ~~in the subjects table matches~~
= faculty_subjects.subject ID ~~in the faculty subjects table~~
where proficiency rating ~~is greater than~~ > 8
and category ID ~~is~~ = 'ART'

SQL

```
SELECT Students.StudFirstName AS FirstName,
   Students.StudLastName AS LastName,
   Student_Schedules.Grade AS Score,
   'Student' AS Type
FROM (((Students INNER JOIN Student_Schedules
ON Students.StudentID =
   Student_Schedules.StudentID)
INNER JOIN Student_Class_Status
```

```
ON Student_Class_Status.ClassStatus =
   Student_Schedules.ClassStatus)
INNER JOIN Classes
ON Classes.ClassID = Student_Schedules.ClassID)
INNER JOIN Subjects
ON Subjects.SubjectID = Classes.SubjectID
WHERE Student_Class_Status.ClassStatusDescription =
   'Completed'
AND Student_Schedules.Grade >= 85
AND Subjects.CategoryID = 'ART'
UNION
SELECT Staff.StfFirstName, Staff.StfLastName,
   Faculty_Subjects.ProficiencyRating AS Score,
   'Faculty' AS Type
FROM (Staff INNER JOIN Faculty_Subjects
ON Staff.StaffID = Faculty_Subjects.StaffID)
INNER JOIN Subjects
ON Subjects.SubjectID = Faculty_Subjects.SubjectID
WHERE Faculty_Subjects.ProficiencyRating > 8
AND Subjects.CategoryID = 'ART'
```

CH10_Good_Art_Students_And_Faculty (16 rows)

FirstName	LastName	Score	Type
Alaina	Hallmark	10	Faculty
Elizabeth	Hallmark	93.27	Student
Kendra	Bonnicksen	88.27	Student
Kendra	Bonnicksen	91.56	Student
Kendra	Bonnicksen	92.05	Student
Liz	Keyser	10	Faculty
Mariya	Sergienko	9	Faculty
Michael	Hernandez	10	Faculty
<< more rows here >>			

Bowling League Database

"List the tourney matches, team names, and team captains for the teams starting on the odd lane together with the tourney matches, team names, and team captains for the teams starting on the even lane, and sort by tournament date and match number."

Translation/ Clean Up

Select tourney location, tourney date, match ID, team name,
captain name ~~and the constant~~ 'Odd Lane'
from ~~the~~ tournaments ~~table~~
join~~ed with the~~ tourney matches ~~table~~
on tournaments.tourney ID ~~in the tournaments table~~ ~~equals~~
= tourney_matches.tourney ID ~~in the tourney matches table,~~
~~then~~ join~~ed with the~~ teams ~~table~~
on tourney_matches.odd lane team ID ~~in the tourney matches table~~ ~~equals~~ = teams.team ID ~~in the teams table,~~
~~and then~~ join~~ed with the~~ bowlers ~~table~~
on teams.captain ID ~~in the teams table~~
~~equals~~ = bowlers.bowler ID ~~in the bowlers table,~~
~~combined with~~ union all
Select tourney location, tourney date, match ID, team name,
captain name ~~and the constant~~ 'Even Lane'
from ~~the~~ tournaments ~~table~~
join~~ed with the~~ tourney matches ~~table~~
on tournaments.tourney ID ~~in the tournaments table~~ ~~equals~~
= tourney_matches.tourney ID ~~in the tourney matches table,~~
~~then~~ join~~ed with the~~ teams ~~table~~
on tourney_matches.even lane team ID
~~in the tourney matches table~~
~~equals~~ = teams.team ID ~~in the teams table,~~
~~and then~~ join~~ed with the~~ bowlers ~~table~~
on teams.captain ID ~~in the teams table~~
~~equals~~ = bowlers.bowler ID ~~in the bowlers table~~,
order by ~~tourney date~~ 2, ~~and match ID~~ 3

```
SQL         SELECT Tournaments.TourneyLocation,
               Tournaments.TourneyDate,
               Tourney_Matches.MatchID, Teams.TeamName,
               Bowlers.BowlerLastName || ', ' ||
               Bowlers.BowlerFirstName AS Captain,
               'Odd Lane' AS Lane
            FROM ((Tournaments INNER JOIN Tourney_Matches
            ON Tournaments.TourneyID =
               Tourney_Matches.TourneyID)
            INNER JOIN Teams
            ON Teams.TeamID =
               Tourney_Matches.OddLaneTeamID)
            INNER JOIN Bowlers
            ON Bowlers.BowlerID = Teams.CaptainID
            UNION ALL
            SELECT Tournaments.TourneyLocation,
               Tournaments.TourneyDate,
               Tourney_Matches.MatchID, Teams.TeamName,
               Bowlers.BowlerLastName || ', ' ||
               Bowlers.BowlerFirstName AS Captain,
               'Even Lane' AS Lane
            FROM ((Tournaments INNER JOIN Tourney_Matches
            ON Tournaments.TourneyID =
               Tourney_Matches.TourneyID)
            INNER JOIN Teams
            ON Teams.TeamID =
               Tourney_Matches.EvenLaneTeamID)
            INNER JOIN Bowlers
            ON Bowlers.BowlerID = Teams.CaptainID
            ORDER BY 2, 3
```

Notice that the two SELECT statements are almost identical! The only difference is the first SELECT statement links Tourney_Matches with Teams on OddLaneTeamID, and the second uses EvenLaneTeamID. Also note that we decided in the final solution to sort by relative column number (the second and third columns) rather than column name (TourneyDate and MatchID). Finally, you can use UNION ALL because a team is never going to compete against itself.

CH10_Bowling_Schedule (114 rows)

Tourney Location	Tourney Date	MatchID	Team Name	Captain	Lane
Red Rooster Lanes	2007-09-04	1	Marlins	Fournier, David	Odd Lane
Red Rooster Lanes	2007-09-04	1	Sharks	Patterson, Ann	Even Lane
Red Rooster Lanes	2007-09-04	2	Barracudas	Sheskey, Richard	Even Lane
Red Rooster Lanes	2007-09-04	2	Terrapins	Viescas, Carol	Odd Lane
Red Rooster Lanes	2007-09-04	3	Dolphins	Viescas, Suzanne	Odd Lane
Red Rooster Lanes	2007-09-04	3	Orcas	Thompson, Sarah	Even Lane
Red Rooster Lanes	2007-09-04	4	Manatees	Viescas, Michael	Odd Lane
Red Rooster Lanes	2007-09-04	4	Swordfish	Rosales, Joe	Even Lane
Thunderbird Lanes	2007-09-11	5	Marlins	Fournier, David	Even Lane
Thunderbird Lanes	2007-09-11	5	Terrapins	Viescas, Carol	Odd Lane
<< more rows here >>					

Recipes Database

"Create an index list of all the recipe classes, recipe titles, and ingredients."

Translation/ Clean Up

Select recipe class description, ~~and the constant~~ 'Recipe Class'
from ~~the~~ recipe classes ~~table~~
~~combined with~~ union
Select recipe title, ~~and the constant~~ 'Recipe'
from ~~the~~ recipes ~~table~~
~~combined with~~ union
Select ingredient name, ~~and the constant~~ 'Ingredient'
from ~~the~~ ingredients ~~table~~

SQL

```
SELECT Recipe_Classes.RecipeClassDescription
   AS IndexName, 'Recipe Class' AS Type
FROM Recipe_Classes
UNION
SELECT Recipes.RecipeTitle, 'Recipe' AS Type
FROM Recipes
UNION
SELECT Ingredients.IngredientName,
   'Ingredient' AS Type
FROM Ingredients
```

CH10_Classes_Recipes_Ingredients (101 rows)

IndexName	Type
Asparagus	Ingredient
Asparagus	Recipe
Bacon	Ingredient
Balsamic vinaigrette dressing	Ingredient
Beef	Ingredient
Beef drippings	Ingredient
Bird's custard powder	Ingredient
Black olives	Ingredient
<< more rows here >>	

SUMMARY

We began the chapter by defining UNION and showing you the difference between linking two tables with a JOIN and combining two tables with a UNION.

We next explained how to construct a simple UNION using two SELECT statements, each of which asked for columns from a single table. We explained the significance of the ALL keyword and recommended that you use it either when you know the queries produce no duplicates or when you don't care. We then progressed to combining two complex SELECT statements that each used a JOIN on multiple tables. Next we showed how to use UNION to combine more than two result sets. We wrapped up our discussion of UNION syntax by showing how to sort the result.

We explained how UNION is useful and listed a variety of requests that you can solve using UNION. The Sample Statements section showed you one or two examples of how to use UNION in each of the sample databases, including the logic behind constructing these requests.

The following section presents a number of requests that you can work out on your own.

Problems for You to Solve

Below, we show you the request statement and the name of the solution query in the sample databases. If you want some practice, you can work out the SQL you need for each request and then check your answer with the query we saved in the samples. Don't worry if your syntax doesn't exactly match the syntax of the queries we saved—as long as your result set is the same.

Sales Orders Database

1. *"List the customers who ordered a helmet together with the vendors who provide helmets."* (Hint: This involves creating a UNION of two complex JOINs.) You can find the solution in CH10_Customer_Helmets_Vendor_Helmets (91 rows).

Entertainment Agency Database

1. *"Display a combined list of customers and entertainers."*
   (Hint: Be careful to create an expression for one of the names so that you have the same number of columns in both SELECT statements.)
   You can find the solution in CH10_Customers_UNION_Entertainers (28 rows).
2. *"Produce a list of customers who like contemporary music together with a list of entertainers who play contemporary music."*
   (Hint: You need to UNION two complex JOINs to solve this one.)
   You can find the solution in CH10_Customers_Entertainers_Contemporary (5 rows).

School Scheduling Database

1. *"Create a mailing list for students and staff, sorted by ZIP Code."*
   (Hint: Try using a relative column number for the sort.)
   You can find the solution in CH10_Student_Staff_Mailing_List (45 rows).

Bowling League Database

1. *"Find the bowlers who had a raw score of 165 or better at Thunderbird Lanes combined with bowlers who had a raw score of 150 or better at Bolero Lanes."*
   (Hint: This is another of those problems that can also be solved with a single SELECT statement and a complex WHERE clause.)
   You can find the solution using UNION in CH10_Good_Bowlers_TBird_Bolero_UNION (129 rows). You can find the solution using WHERE in CH10_Good_Bowlers_TBird_ Bolero_WHERE (135 rows).

2. Can you explain why the row counts are different in the previous solution queries?
   (Hint: Try using UNION ALL in the first query.)

Recipes Database

1. *"Display a list of all ingredients and their default measurement amounts together with ingredients used in recipes and the measurement amount for each recipe."*
   (Hint: You need one simple JOIN and one complex JOIN to solve this.)
   You can find the solution in CH10_Ingredient_Recipe_Measurements (144 rows).

This page intentionally left blank

11

Subqueries

"We can't solve problems by using the same kind of thinking we used when we created them."
—Albert Einstein

Topics Covered in This Chapter

What Is a Subquery?

Subqueries as Column Expressions

Subqueries as Filters

Uses for Subqueries

Sample Statements

Summary

Problems for You to Solve

In the previous three chapters, we showed you many ways to work with data from more than one table. All the techniques we've covered to this point have been focused on linking subsets of information—one or more columns and one or more rows from an entire table or a query embedded in the FROM clause. We've also explored combining sets of information using the UNION operator. In this chapter, we'll show you effective ways to fetch a single column from a table or query and use it as a value expression in either a SELECT clause or a WHERE clause.

There are two significant points you should learn in this chapter.

1. There's always more than one way to solve a particular problem in SQL. In fact, this chapter will show you new ways to solve problems already covered in previous chapters.

2. You can build complex filters that do not rely on the tables in your FROM clause. This is an important concept because using a subquery in a WHERE clause is the only way to get the correct number of rows in your answer when you want rows from one table based on the filtered contents from other related tables. We'll explain this in more detail later in the chapter.

❖ **Note** This chapter covers advanced concepts and assumes that you've read and thoroughly understood Chapter 7, Thinking in Sets; Chapter 8, INNER JOINs; and Chapter 9, OUTER JOINs.

What Is a Subquery?

Simply put, a *subquery* is a SELECT expression that you embed inside one of the clauses of a SELECT statement to form your final query statement. In this chapter, we'll define more formally a subquery and show how to use it other than in the FROM clause.

The SQL Standard defines three types of subqueries:

1. **Row subquery**—an embedded SELECT expression that returns more than one column and no more than one row
2. **Table subquery**—an embedded SELECT expression that returns one or more columns and zero to many rows
3. **Scalar subquery**—an embedded SELECT expression that returns only one column and no more than one row

Row Subqueries

You've already created queries that embed a SELECT statement in a FROM clause to let you filter rows before joining that result with other tables or queries. (That's called a table subquery, as you'll learn below.) A *row subquery* is a special form of a SELECT statement that returns more than one column but only one row.

In the SQL Standard, you can use a row subquery to build something the standard calls a *row value constructor*. When you create a WHERE clause, you build a search condition that is typically some sort of comparison of one column from one of your tables either with another column or with a literal. The SQL Standard, however, allows you to build a search condition that compares multiple values as a logical row with another set of values as a logical row

(two row value constructors). You can enter the list of comparison values either by making a list in parentheses or by using a row subquery to fetch a single row from one of your tables. The bad news is that not many commercial database systems support this syntax.

Why might this be useful? Consider a Products table that has a compound part identifier in two separate fields. The first part of the identifier might be characters that indicate the subclass of parts (SKUClass), such as CPU or DSK for a computer parts manufacturer. The second part of the identifier could be a number that identifies the part within the subclass (SKUNumber). Let's say you want all parts that have a combined identifier of DSK09775 or higher. Here's an example of a WHERE clause that uses a row value constructor to solve the problem.

```
SQL         SELECT SKUClass, SKUNumber, ProductName
            FROM Products
            WHERE
               (SKUClass, SKUNumber)
               >= ('DSK', 9775)
```

The preceding WHERE clause asks for rows where the combination of SKUClass and SKUNumber is greater than the combination of DSK and 9775. It's the same as requesting the following.

```
SQL         SELECT SKUClass, SKUNumber, ProductName
            FROM Products
            WHERE (SKUClass > 'DSK')
            OR ((SKUClass = 'DSK')
            AND (SKUNumber >= 9775))
```

Here's where you could substitute a SELECT statement that returns a single row of two columns—a row subquery—for the second part of the comparison (probably using a WHERE clause to limit the result to one row). Most commercial databases support neither a row value constructor nor row subqueries. That's all we're going to say about them in this chapter.

Table Subqueries

Wait a minute! Didn't we already show you how to embed a SELECT expression returning multiple rows and columns inside a FROM clause in the previous three chapters? The answer is yes—we snuck it in on you! We've already liberally used table subqueries in the previous chapters to specify a complex

result that we then embedded in the FROM clause of another query. In this chapter, we'll show you how to use a table subquery as the source for the list of comparison values for an IN predicate—something about which you learned the basics in Chapter 6, Filtering Your Data. We'll also teach you a few new comparison predicate keywords that are used only with table subqueries.

Scalar Subqueries

In this chapter we'll also show how to use a scalar subquery anywhere you might otherwise use a value expression. A scalar subquery lets you fetch a single column or calculated expression from another table that does not have to be in the FROM clause of the main query. You can use the single value fetched by a scalar subquery in the list of columns you request in a SELECT clause or as a comparison value in a WHERE clause.

Subqueries as Column Expressions

In Chapter 5, Getting More Than Simple Columns, you learned a lot about using expressions to generate calculated columns to be output by your query. We didn't tell you then that you can also use a special type of SELECT statement—a subquery—to fetch data from another table, even if the table isn't in your FROM clause.

Syntax

Let's go back to the basics and take a look at a simple form of a SELECT statement in Figure 11–1.

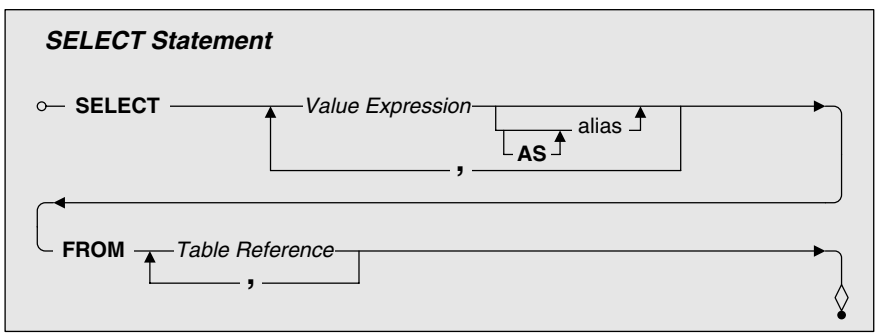


Figure 11–1 *The syntax diagram for a simple SELECT statement*

This looks simple, but it really isn't! In fact, the value expression part can be quite complex. Figure 11–2 shows all the options that can constitute a value expression.

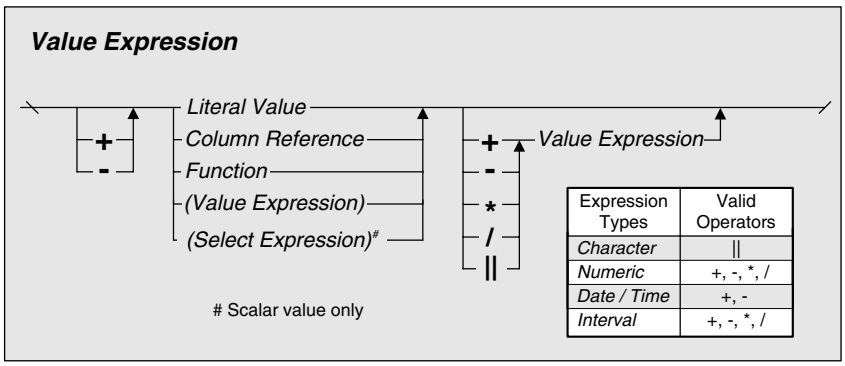


Figure 11–2 *The syntax diagram for a value expression*

In Chapter 5, we showed you how to create basic value expressions using literal values, column references, and functions. Notice that *SELECT expression* now appears on the list. This means that you can embed a scalar subquery in the list of expressions immediately following the SELECT keyword. As noted earlier, a scalar subquery is a SELECT expression that returns exactly one column and no more than one row. This makes sense because you're substituting the subquery where you would normally enter a single column name or expression that results in a single column.

You might be wondering at this point, "Why is this useful?" A subquery used in this way lets you pluck a single value from some other table or query to include in the output of your query. You don't need to reference the table or query that is the source of the data in the FROM clause of the subquery at all in the FROM clause of the outer query. In most cases, you will need to add criteria in the WHERE clause of the subquery to make certain it returns no more than one row. You can even have the criteria in the subquery reference a value being returned by the outer query to pluck out the data related to the current row.

Let's look at some simple examples using only the Customers and Orders tables from the Sales Orders example database. Figure 11–3 shows the relationship between these two tables.

Now, let's build a query that lists all the orders for a particular date and plucks the related customer last name from the Customers table using a subquery.

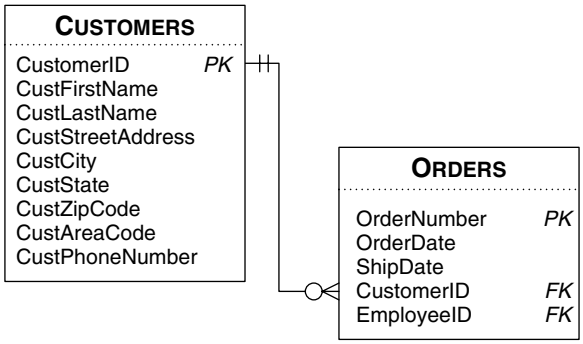


Figure 11–3 *The Customers and Orders tables*

❖ **Note** Throughout this chapter, we use the "Request/Translation/Clean Up/SQL" technique introduced in Chapter 4, Creating a Simple Query. In addition, we include parentheses around the parts that are subqueries in the Clean Up step and indent the subqueries where possible to help you see how we are using them.

"Show me all the orders shipped on October 3, 2007, and each order's related customer last name."

Translation Select order number, order date, ship date, and also select the related customer last name out of the customers table from the orders table where ship date is October 3, 2007

Clean Up Select order number, order date, ship date,
~~and also~~ (select ~~the related~~ customer last name ~~out of the~~
from customers ~~table~~)
from ~~the~~ orders ~~table~~
where ship date ~~is~~ = ~~October 3, 2007~~ '2007-10-03'

SQL
```
SELECT Orders.OrderNumber, Orders.OrderDate,
   Orders.ShipDate,
   (SELECT Customers.CustLastName
   FROM Customers
   WHERE Customers.CustomerID =
      Orders.CustomerID)
FROM Orders
WHERE Orders.ShipDate = '2007-10-03'
```

Notice that we had to restrict the value of the CustomerID in the subquery to the value of the CustomerID in each row we're fetching from the Orders table. Otherwise, we'll get *all* the rows in Customers in the subquery. Remem-

ber that this must be a scalar subquery—a query that returns only one value from one row—so we must do something to restrict what gets returned to no more than one row. Because CustomerID is the primary key of the Customers table, we can be confident that the match on the CustomerID column from the Orders table will return exactly one row.

Those of you who really caught on to the concept of INNER JOIN in Chapter 8 are probably wondering why you would want to solve this problem as just described rather than to JOIN Orders to Customers in the FROM clause of the outer query. Right now we're focusing on the *concept* of using a subquery to create an output column with a very simple example. In truth, you probably should solve this particular problem with the following query using an INNER JOIN.

```
SQL         SELECT Orders.OrderNumber, Orders.OrderDate,
               Orders.ShipDate, Customers.CustLastName
            FROM Customers
            INNER JOIN Orders
            ON Customers.CustomerID = Orders.OrderID
            WHERE Orders.ShipDate = '2007-10-03'
```

An Introduction to Aggregate Functions: COUNT and MAX

Now that you understand the basic concept of using a subquery to generate an output column, let's expand your horizons and see how this feature can be really useful. First, we need to give you an overview of a couple of aggregate functions. (We'll cover all the aggregate functions in detail in the next chapter.)

The SQL Standard defines many functions that calculate values in a query. One subclass of functions—aggregate functions—lets you calculate a single value for a group of rows in a result set. For example, you can use an aggregate function to count the rows, find the largest or smallest value within the set of rows, or calculate the average or total of some value or expression across the result set.

Let's take a look at a couple of these functions and then see how they can be most useful in a subquery. Figure 11–4 (on page 376) shows the diagram for the COUNT and MAX functions that can generate an output column in a SELECT clause.

You can use COUNT to determine the number of rows or non-Null values in a result set. Use COUNT(*) to find out how many rows are in the entire set. If

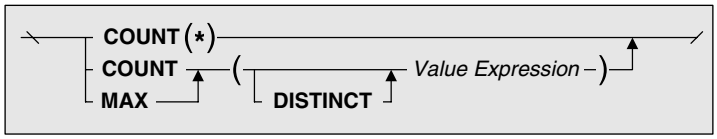


Figure 11–4 *Using the COUNT and MAX aggregate functions*

you specify a particular column in the result set using COUNT(*column_name*), the database system counts the number of rows with non-Null values in that column. You can also ask to count only the unique values by adding the DISTINCT keyword.

Likewise, you can find the largest value in a column by using MAX. If the value expression is numeric, you get the largest number value from the column or expression you specify. If the value expression returns a character data type, the largest value will depend on the collating sequence of your database system.

Let's use these functions in a subquery to solve a couple of interesting problems.

"List all the customer names and a count of the orders they placed."

Translation: Select customer first name, customer last name, and also select the count of orders from the orders table for this customer from the customers table

Clean Up: Select customer first name, customer last name, ~~and also~~ (select ~~the~~ count ~~of orders~~ (*) from ~~the~~ orders ~~table for this customer~~ where orders.customer ID = customers.customer ID) from ~~the~~ customers ~~table~~

SQL:

```
SELECT Customers.CustFirstName,
   Customers.CustLastName,
   (SELECT COUNT(*)
   FROM Orders
   WHERE Orders.CustomerID =
      Customers.CustomerID)
AS CountOfOrders
FROM Customers
```

Subqueries as output columns are starting to look interesting now! In Part IV, Summarizing and Grouping Data, you'll learn more about creative ways to use aggregate functions. But if all you want is a count of related rows, a subquery is a good way to do it. In fact, if you don't want anything other than the cus-

tomer name and the count of orders, this is just about the only way to solve the problem. If you add the Orders table to the FROM clause of the main query (`FROM Customers INNER JOIN Orders ON Customers.CustomerID = Orders.CustomerID`), you'll get multiple rows for each customer who placed more than one order. In Chapter 13, Grouping Data, you'll learn about another way that involves grouping the rows on customer name.

Let's look at an interesting problem that takes advantage of another aggregate function—MAX.

"Show me a list of customers and the last date on which they placed an order."

Translation
Select customer first name, customer last name, and also select the highest order date from the orders table for this customer from the customers table

Clean Up
Select customer first name, customer last name, ~~and also~~
(select ~~the highest~~ max(order date)
from ~~the~~ orders ~~table~~
~~for this customer~~ where orders.customer ID =
customers.customer ID)
from ~~the~~ customers ~~table~~

SQL

```
SELECT Customers.CustFirstName,
   Customers.CustLastName,
   (SELECT MAX(OrderDate)
   FROM Orders
   WHERE Orders.CustomerID =
      Customers.CustomerID)
AS LastOrderDate
FROM Customers
```

As you can imagine, using MAX in this way works well for finding the highest or most recent value from any related table. We'll show you a number of other ways to use these functions in the Sample Statements section later in this chapter.

Subqueries as Filters

In Chapter 6, you learned how to filter the information retrieved by adding a WHERE clause. You also learned how to use both simple and complex comparisons to get only the rows you want in your result set. Now we'll build on your skills and show you how to use a subquery as one of the comparison arguments to do more sophisticated filtering.

Syntax

Let's revisit the SELECT statement from Figure 11–1 and look at the syntax for building a query with a simple comparison predicate in a WHERE clause. Figure 11–5 shows the simplified diagram.

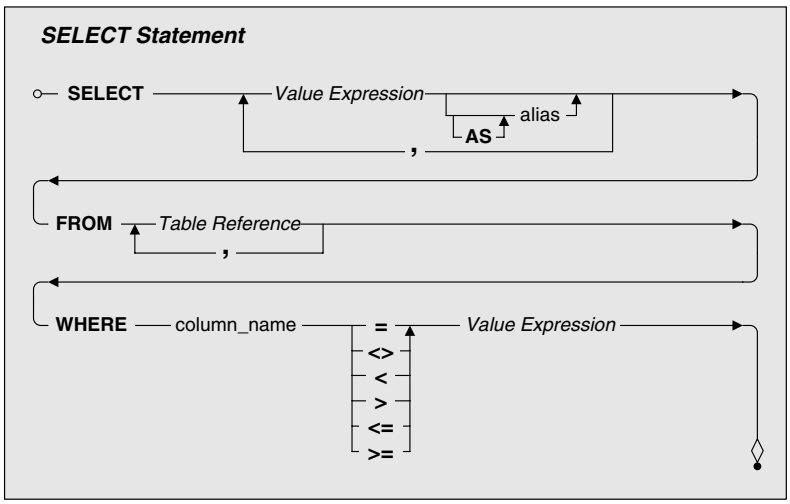


Figure 11–5 *Filtering a result using a simple comparison predicate*

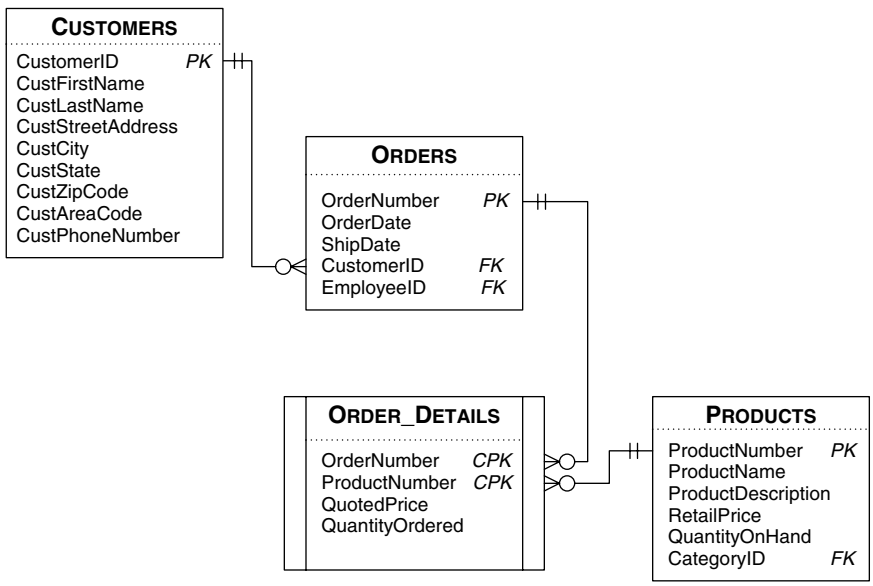


Figure 11–6 *The tables required to list all the details about an order*

As you remember from Figure 11–2, a value expression can be a subquery. In the simple example in Figure 11–5, you're comparing the value expression to a single column. Thus, the value expression must be a single value—that is, a scalar subquery that returns exactly one column and no more than one row. Let's solve a simple problem requiring a comparison to a value returned from a subquery. In this example, we are going to ask for all the details about customer orders, but we want only the *last* order for each customer. Figure 11–6 shows the tables needed.

"List customers and all the details from their last order."

Translation
Select customer first name, customer last name, order number, order date, product number, product name, and quantity ordered from the customers table joined with the orders table on customer ID in the customers table equals customer ID in the orders table, then joined with the order details table on order number in the orders table equals order number in the order details table, and then joined with the products table on product number in the products table equals product number in the order details table where the order date equals the maximum order date from the orders table for this customer

Clean Up
Select customer first name, customer last name, order number, order date, product number, product name, ~~and~~ quantity ordered from ~~the~~ customers ~~table~~ inner join~~ed with the~~ orders ~~table~~ on customers.customer ID ~~in the customers table equals~~ = orders.customer ID ~~in the orders table, then~~ inner join~~ed with the~~ order details ~~table~~ on orders.order number ~~in the orders table equals~~ = order_details.order number ~~in the order details table, and then~~ inner join~~ed with the~~ products ~~table~~ on products.product number ~~in the products table equals~~ = order_details.product number ~~in the order details table~~ where ~~the~~ order date ~~equals~~ =

(select ~~the~~ max~~imum~~ (order date) from ~~the~~ orders ~~table for this customer~~ where orders.customer ID
= customers.customer ID)

SQL
```
SELECT Customers.CustFirstName,
   Customers.CustLastName, Orders.OrderNumber,
   Orders.OrderDate,
   Order_Details.ProductNumber,
   Products.ProductName,
   Order_Details.QuantityOrdered
FROM ((Customers
INNER JOIN Orders
ON Customers.CustomerID = Orders.CustomerID)
INNER JOIN Order_Details
ON Orders.OrderNumber = Order_Details.OrderNumber)
INNER JOIN Products
ON Products.ProductNumber =
   Order_Details.ProductNumber
WHERE Orders.OrderDate =
   (SELECT MAX(OrderDate)
   FROM Orders AS O2
   WHERE O2.CustomerID = Customers.CustomerID)
```

Did you notice that we gave an alias name to the second reference to the Orders table (that is, the Orders table in the subquery)? Even if you leave out the alias name, many database systems will recognize that you mean the copy of the Orders table within the subquery. In fact, the SQL Standard dictates that any unqualified reference should be resolved from the innermost query first. Still, we added the alias reference to make it crystal clear that the copy of the Orders table we're referencing in the WHERE clause of the subquery is the one in the FROM clause of the subquery. If you follow this practice, your request will be much easier to understand—either by you when you come back to it some months later or by someone else who has to figure out what your request meant.

Special Predicate Keywords for Subqueries

The SQL Standard defines a number of special predicate keywords for use in a WHERE clause with a subquery.

Set Membership: IN

You learned in Chapter 6 how to use the IN keyword in a WHERE clause to compare a column or expression to a list of values. You now know that each value expression in the IN list *could* be a scalar subquery. How about using a

subquery to generate the entire list? As Figure 11–7 shows, you can certainly do that!

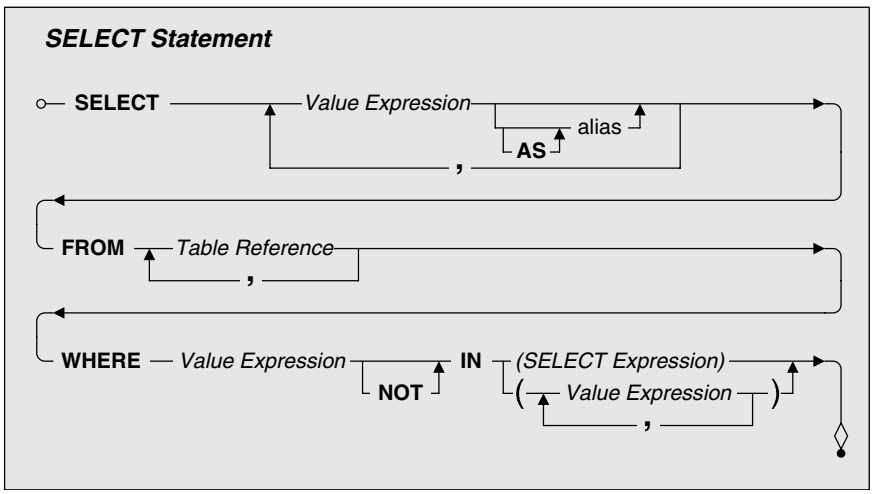


Figure 11–7 *Using a subquery with an IN predicate*

In this case, you can use a table subquery that returns one column and as many rows as necessary to build the list. Let's use the Recipes sample database for an example. Figure 11–8 shows the tables of interest.

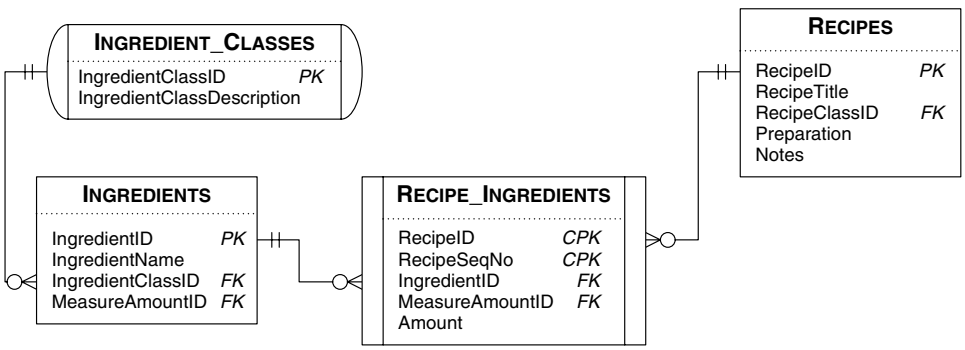


Figure 11–8 *The tables needed to list recipes and their ingredients*

Let's suppose you're having someone over for dinner who just adores seafood. Although you know you have a number of recipes containing seafood ingredients, you're not sure of all the ingredient names in your database. You do know that you have an IngredientClassDescription of Seafood,

so you can join all the tables and filter on IngredientClassDescription—or you can get creative and use subqueries and the IN predicate instead.

"List all my recipes that have a seafood ingredient."

Translation
Select recipe title from the recipes table where the recipe ID is in the selection of recipe IDs from the recipe ingredients table where the ingredient ID is in the selection of ingredient IDs from the ingredients table joined with the ingredient classes table on ingredient class ID in the ingredients table matches ingredient class ID in the ingredient classes table where ingredient class description is 'seafood'

Clean Up
Select recipe title from ~~the~~ recipes ~~table~~
where ~~the~~ recipe ID ~~is~~ in ~~the~~
(select~~ion of~~ recipe ID~~s~~
from ~~the~~ recipe ingredients ~~table~~
where ~~the~~ ingredient ID ~~is~~ in ~~the~~
(select~~ion of~~ ingredient ID~~s~~
from ~~the~~ ingredients ~~table~~
inner join~~ed with the~~ ingredient classes ~~table~~
on ingredients.ingredient class ID
~~in the ingredients table matches~~
= ingredient_classes.ingredient class ID
~~in the ingredient classes table~~
where ingredient class description ~~is~~ = 'seafood'))

SQL

```
SELECT RecipeTitle
FROM Recipes
WHERE Recipes.RecipeID IN
   (SELECT RecipeID
   FROM Recipe_Ingredients
   WHERE Recipe_Ingredients.IngredientID IN
      (SELECT IngredientID
      FROM Ingredients
      INNER JOIN Ingredient_Classes
      ON Ingredients.IngredientClassID =
         Ingredient_Classes.IngredientClassID
      WHERE
      Ingredient_Classes.IngredientClassDescription
         = 'Seafood'))
```

Did it occur to you that you could put a subquery within a subquery? We actually could have gone one level deeper by eliminating the INNER JOIN from the second subquery. We could have stated the second subquery using the following syntax.

```
SQL         (SELECT IngredientID
            FROM Ingredients
            WHERE Ingredients.IngredientClassID IN
               (SELECT IngredientClassID
               FROM Ingredient_Classes
               WHERE
               Ingredient_Classes.IngredientClassDescription
                  = 'Seafood'))
```

That would be overkill, however, because embedding IN clauses within IN clauses only makes the query harder to read. We did so in the previous example to show you that you *can* do it. It's worth restating, though, that just because you *can* do something doesn't mean you *should!* We think you'll agree that it's easier to see what's going on by using a single IN predicate and a more complex JOIN in the subquery. Here's another solution using this technique.

```
SQL         SELECT RecipeTitle
            FROM Recipes
            WHERE Recipes.RecipeID IN
               (SELECT RecipeID
               FROM (Recipe_Ingredients
               INNER JOIN Ingredients
               ON Recipe_Ingredients.IngredientID =
                  Ingredients.IngredientID)
               INNER JOIN Ingredient_Classes
               ON Ingredients.IngredientClassID =
                  Ingredient_Classes.IngredientClassID
               WHERE
               Ingredient_Classes.IngredientClassDescription
                  = 'Seafood')
```

You might be asking at this point, "Why go to all this trouble? Why not just do the complex JOIN in the outer query and be done with it?" The reason is that you'll get the wrong answer! Actually, the rows returned will all be rows from the Recipes table for seafood recipes, but you might get some rows more than once. Let's try to solve this without the subquery to see why you get duplicate rows.

```
SQL         SELECT RecipeTitle
            FROM ((Recipes
            INNER JOIN Recipe_Ingredients
            ON Recipes.RecipeID =
               Recipe_Ingredients.RecipeID)
```

```
INNER JOIN Ingredients
ON Recipe_Ingredients.IngredientID =
   Ingredients.IngredientID)
INNER JOIN Ingredient_Classes
ON Ingredients.IngredientClassID =
Ingredient_Classes.IngredientClassID
WHERE
Ingredient_Classes.IngredientClassDescription
   = 'Seafood')
```

If you look back at Figure 11–8, you can see that the Recipe_Ingredients table might have many rows for each row in the Recipes table. The result set defined by the FROM clause will contain at least as many rows as there are in Recipe_Ingredients, with the RecipeTitle column value repeated many times. Even when we add the filter to restrict the result to ingredients in class Seafood, we will still get more than one row per recipe in any recipe that has more than one seafood ingredient.

Yes, you could include the DISTINCT keyword, but the odds are your database system will have to do more work to eliminate the duplicates. If you save this as a view using DISTINCT and then try to update data in the view, you'll find that the view is not updatable because DISTINCT masks the unique identity of each underlying row, and your database system won't know which row to update.

Using this subquery technique also becomes really important when you want to list more than just the recipe title. For example, suppose you also want to list *all* the ingredients from *any* recipe that has a seafood ingredient. If you use a complex JOIN in the outer query and filter for an ingredient class of Seafood as we just did, all you will get is seafood ingredients—you won't get all the other ingredients for the recipes. Let's ask one additional and slightly more complex request.

"List recipes and all ingredients for each recipe for recipes that have a seafood ingredient."

Translation Select recipe title and ingredient name from the recipes table joined with the recipe ingredients table on recipe ID in the recipes table equals recipe ID in the recipe ingredients table, and then joined with the ingredients table on ingredient ID in the ingredients table equals ingredient ID in the recipe ingredients table where the recipe ID is in the selection of recipe IDs from the recipe ingredients table joined with the ingredients table on ingredient ID in the recipe ingredients table equals

ingredient ID in the ingredients table, and then joined with the ingredient classes table on ingredient class ID in the ingredients table equals ingredient class ID in the ingredient classes table where ingredient class description is 'seafood'

Clean Up

Select recipe title, ~~and~~ ingredient name
from ~~the~~ recipes ~~table~~
inner join~~ed with the~~ recipe ingredients ~~table~~
on recipes.recipe ID ~~in the recipes table equals~~
= recipe_ingredients.recipe ID ~~in the recipe ingredients table, and then~~ inner join~~ed with the~~ ingredients ~~table~~
on ingredients.ingredient ID ~~in the ingredients table equals~~
= recipe_ingredients.ingredient ID
~~in the recipe ingredients table~~
where ~~the~~ recipe ID ~~is~~ in ~~the~~
(select~~ion of~~ recipe ID~~s~~
from ~~the~~ recipe ingredients ~~table~~
inner join~~ed with the~~ ingredients ~~table~~
on recipe_ingredients.ingredient ID
~~in the recipe ingredients table equals~~
= ingredients.ingredient ID ~~in the ingredients table, and then~~
inner join~~ed with the~~ ingredient classes ~~table~~
on ingredients.ingredient class ID
~~in the ingredients table equals~~
= ingredient_classes.ingredient class ID
~~in the ingredient classes table~~
where ingredient class description ~~is~~ = 'seafood')

SQL

```
SELECT Recipes.RecipeTitle,
   Ingredients.IngredientName
FROM (Recipes
INNER JOIN Recipe_Ingredients
ON Recipes.RecipeID =
   Recipe_Ingredients.RecipeID)
INNER JOIN Ingredients
ON Ingredients.IngredientID =
   Recipe_Ingredients.IngredientID
WHERE Recipes.RecipeID IN
   (SELECT RecipeID
   FROM (Recipe_Ingredients
```

```
INNER JOIN Ingredients
ON Recipe_Ingredients.IngredientID =
   Ingredients.IngredientID)
INNER JOIN Ingredient_Classes
ON Ingredients.IngredientClassID =
   Ingredient_Classes.IngredientClassID
WHERE
Ingredient_Classes.IngredientClassDescription
   = 'Seafood')
```

The key here is that the complex INNER JOIN in the main part of the query retrieves *all* the ingredients for the recipes selected, and the complex subquery returns a list of recipe IDs for just the seafood recipes. It seems like we're doing a complex JOIN twice, but there's method in the madness!

Quantified: ALL, SOME, and ANY

As you have just seen, the IN predicate lets you compare a column or expression to a list to see whether that column or expression is *in* the list. In other words, the column or expression *equals* one of the members of the list. If you want to find out whether the column or expression is greater than or less than any, all, or some of the items in the list, you can use a *quantified predicate*. Figure 11–9 shows the syntax.

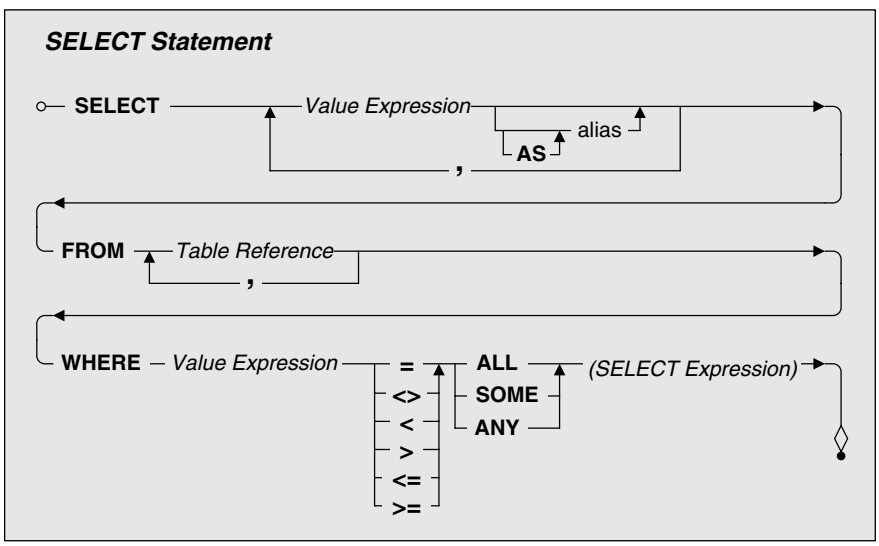


Figure 11–9 *Using a quantified predicate in a SELECT statement*

In this case, the SELECT expression must be a table subquery that returns exactly one column and zero or more rows. When the subquery returns more than one row, the values in the rows make up a list. As you can see, this predicate combines a comparison operator with a keyword that tells your database system how to apply the operator to the members of the list. When you use the keyword ALL, the comparison must be true for all the values returned by the subquery. When you use either of the keywords SOME or ANY, the comparison need be true for only one value in the list.

If you think about it, when the subquery returns multiple rows, asking for = ALL will always be false unless all the values returned by the subquery are the same and the value expression on the left of the comparison equals all of them. By the same logic, you might think that <> ANY will always be false if the value expression on the left *does* equal any of the values in the list. In truth, the SQL Standard treats SOME and ANY the same. So if you say <> SOME or <> ANY, then the predicate is true if the value expression on the left does not equal at least one of the values in the list. Another confusing point is that if the subquery returns no rows, then any comparison predicate with the ALL keyword is true, and any comparison predicate with the SOME or ANY keyword is false.

Let's work through a couple of requests to see quantified predicates in action. First, let's do a problem in the Recipes database. Refer to Figure 11–8 to see the tables we'll use.

"Show me the recipes that have beef or garlic."

Translation: Select recipe title from the recipes table where recipe ID is in the selection of recipe IDs from the recipe ingredients table where ingredient ID equals any of the selection of ingredient IDs from the ingredients table where ingredient name is 'beef' or 'garlic'

Clean Up:
Select recipe title from ~~the~~ recipes ~~table~~
where recipe ID ~~is~~ in ~~the~~
(select~~ion of~~ recipe ID~~s~~ from ~~the~~ recipe ingredients ~~table~~
where ingredient ID ~~equals~~ = any ~~of the~~
(select~~ion of~~ ingredient ID~~s~~ from ~~the~~ ingredients ~~table~~
where ingredient name ~~is~~ in 'beef' ~~or~~ 'garlic'))

SQL:

```
SELECT Recipes.RecipeTitle
FROM Recipes
WHERE Recipes.RecipeID IN
   (SELECT Recipe_Ingredients.RecipeID
   FROM Recipe_Ingredients
```

```
WHERE Recipe_Ingredients.IngredientID = ANY
   (SELECT Ingredients.IngredientID
   FROM Ingredients
   WHERE Ingredients.IngredientName
   IN ('Beef', 'Garlic')))
```

Do you get the feeling we could have also used IN instead of = ANY? If so, you're right! We could have also created a JOIN between Recipe_Ingredients and Ingredients in the first subquery to return the requisite list of RecipeIDs. As we stated at the beginning of the chapter, there's almost always more than one way to solve a particular problem in SQL. Sometimes, using a quantified predicate might make your request clearer.

Let's now solve a more complex problem to show you the real power of quantified predicates. This example uses the Sales Orders sample database, and Figure 11–10 shows the tables involved.

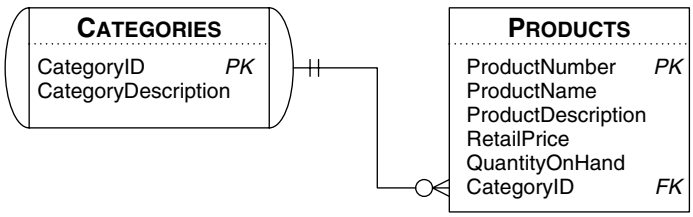


Figure 11–10 *The relationship of the Categories and Products tables*

"Find all accessories that are priced greater than any clothing item."

Translation Select product name and retail price from the products table joined with the categories table on category ID in the products table matches category ID in the categories table where category description is 'accessories' and retail price is greater than all the selection of retail price from the products table joined with the categories table on category ID in the products table matches category ID in the categories table where category name is 'clothing'

Clean Up Select product name ~~and~~ retail price
from ~~the~~ products ~~table~~
inner join~~ed with the~~ categories ~~table~~
on products.category ID ~~in the products table~~
~~matches~~ = categories.category ID ~~in the categories table~~
where category description ~~is~~ = 'accessories'
and retail price ~~is greater than~~ > all ~~the~~

(selection of retail price
from the products table
inner joined with the categories table
on products.category ID in the products table
matches = categories.category ID in the categories table
where category name is = 'clothing')

SQL

```
SELECT Products.ProductName,
   Products.RetailPrice
FROM Products
INNER JOIN Categories
ON Products.CategoryID
   = Categories.CategoryID
WHERE Categories.CategoryDescription =
   'Accessories'
AND Products.RetailPrice > ALL
   (SELECT Products.RetailPrice
   FROM Products
   INNER JOIN Categories
   ON Products.CategoryID =
      Categories.CategoryID
   WHERE Categories.CategoryDescription =
      'Clothing')
```

What's happening here? The subquery fetches all the prices for clothing items. The outer query then lists all accessories whose prices are greater than *all* the prices in the clothing items subquery. Note that you could also solve this query by finding the RetailPrice that is greater than the MAX price fetched in a subquery, but the point here is to demonstrate a use of ALL.

Existence: EXISTS

Both set membership (IN) and quantified (SOME, ANY, ALL) predicates perform a comparison with a value expression—usually a column from the source you specify in the FROM clause of your outer query. Sometimes it's useful to know simply that a related row EXISTS in the result set returned by a subquery. In Chapter 8, we showed a technique for solving AND problems using complex INNER JOINs. You can also use EXISTS to solve those same sorts of problems. Let's take another look at a problem we solved in Chapter 8.

"Find all the customers who ordered a bicycle and also ordered a helmet."

Translation Select customer ID, customer first name, and customer last name from the customers table where there exists some row from the orders table joined with the order details table on

order ID in the orders table equals order ID in the order details table, and then joined with the products table on product ID in the products table equals product ID in the order details table where product name contains 'Bike' and the orders table customer ID equals the customers table customer ID, and there also exists some row from the orders table joined with the order details table on order ID in the orders table equals order ID in the order details table, and then joined with the products table on product ID in the products table equals product ID in the order details table where product name contains 'Helmet' and the orders table customer ID equals the customers table customer ID

Clean Up

Select customer ID, customer first name, ~~and~~ customer last name
from ~~the~~ customers ~~table~~
where ~~there~~ exists ~~some row~~
(select * from ~~the~~ orders ~~table~~
inner join~~ed with the~~ order details ~~table~~
on orders.order ID ~~in the orders table equals~~
= order_details.order ID ~~in the order details table, and then~~
inner join~~ed with th~~e products ~~table~~
on products.product ID ~~in the products table equals~~
= order_details.product ID ~~in the order details table~~
where product name ~~contains~~ LIKE '%Bike'
and ~~the~~ orders ~~table~~ customer ID ~~equals~~ =
~~the~~ customers ~~table~~ customer ID)~~,~~
and ~~there also~~ exists ~~some row~~
(select * from ~~the~~ orders ~~table~~
inner join~~ed with the~~ order details ~~table~~
on orders.order ID ~~in the orders table equals~~
= order_details.order ID ~~in the order details table, and then~~
inner join~~ed with the~~ products ~~table~~
on products.product ID ~~in the products table equals~~
= order_details.product ID ~~in the order details table~~
where product name ~~contains~~ LIKE '%Helmet'
and ~~the~~ orders ~~table~~ customer ID ~~equals~~ =
~~the~~ customers ~~table~~ customer ID)

SQL

```
SELECT Customers.CustomerID,
   Customers.CustFirstName,
   Customers.CustLastName
FROM Customers
WHERE EXISTS
```

```
    (SELECT *
    FROM (Orders
    INNER JOIN Order_Details
    ON Orders.OrderNumber =
       Order_Details.OrderNumber)
    INNER JOIN Products
    ON Products.ProductNumber =
       Order_Details.ProductNumber
    WHERE Products.ProductName LIKE '%Bike'
    AND Orders.CustomerID =
       Customers.CustomerID)
AND EXISTS
    (SELECT *
    FROM (Orders
    INNER JOIN Order_Details
    ON Orders.OrderNumber =
       Order_Details.OrderNumber)
    INNER JOIN Products
    ON Products.ProductNumber =
       Order_Details.ProductNumber
    WHERE Products.ProductName LIKE '%Helmet'
    AND Orders.CustomerID =
       Customers.CustomerID)
```

❖ **Note** We know that all the bicycle products in the sample database contain the word *Bike* at the end of the product name. There is also a product in the database with a product name of Viscount C-500 Wireless Bike Computer, but that isn't a bicycle. Likewise, we know that all the helmet products in the sample database contain the word *Helmet* at the end of the product name, and another product name, Dog Ear Helmet Mount Mirrors, has *Helmet* in the middle but is not a helmet.

For this reason, we chose to use LIKE '%Bike' and LIKE '%Helmet' rather than LIKE '%Bike%' and LIKE '%Helmet%' to ensure that we're selecting the correct products. In truth, the table should probably have been designed with a product category field to make the filtering of the data more precise. As we pointed out in Chapter 2, Ensuring Your Database Structure Is Sound, little design problems like this can make getting the right answer more difficult.

Notice that you can use any column name from any of the tables in the FROM clause as the column to be fetched in the SELECT clause of the subquery. We chose to use the shorthand "*" for all columns. Stated another way, this query is asking, "Give me the customers for whom there exists some row in order details for a bike and for which there also exists some row in order details for

a helmet." Because we didn't match on the OrderID column, we don't care if the customer ordered a bike in one order and a helmet in another one.

❖ **Note** Because this is such an interesting query, we saved this solution as Cust_Bikes_And_Helmets_EXISTS in the sample database. You can find the original INNER JOIN solution saved as Cust_Bike_And_Helmets_JOIN.

Uses for Subqueries

At this point, you should have a pretty good understanding of the concept of using a subquery either to generate an output column or to perform a complex comparison in a WHERE clause. The best way to give you an idea of the wide range of uses for subqueries is to list some problems you can solve with subqueries and then present a robust set of examples in the Sample Statements section.

Build Subqueries as Column Expressions

As mentioned earlier in this chapter, using a subquery to fetch a single value from a related table is probably more effectively done with a JOIN. When you consider aggregate functions, however, subqueries to fetch the result of a function calculation make the idea much more interesting. We'll explore this use of aggregate functions further in the next chapter. In the meantime, here are some problems you can solve using a subquery to generate an output column.

"List vendors and a count of the products they sell to us."
"Display products and the latest date the product was ordered."
"Show me entertainers and the count of each entertainer's engagements."
"Display all customers and the date of the last booking each made."
"List all staff members and the count of classes each teaches."
"Display all subjects and the count of classes for each subject on Monday."
"Show me all the bowlers and a count of games each bowled."
"Display the bowlers and the highest game each bowled."
"List all the meats and the count of recipes each appears in."
"Show me the types of recipes and the count of recipes in each type."

Use Subqueries as Filters

Now that you know about subqueries, you can really expand your kit of tools for solving complex queries. In this chapter, we explored many interesting ways to use subqueries as filters in a WHERE clause. In Chapter 14, Filtering Grouped Data, we'll show you how to use subqueries as filters for groups of information in a HAVING clause.

Here's a sample of problems you can solve using subqueries as a filter for rows in a WHERE clause. Note that we solved many of these same problems in earlier chapters. Now, you get to think about solving them an alternate way by using a subquery!

❖ **Note** As a hint, we've included the keyword(s) you can use to solve the problem in parentheses after the problem statement.

"List customers who ordered bikes." (IN)

"Display customers who ordered clothing or accessories." (= SOME)

"Find all the customers who ever ordered a bicycle helmet." (IN)

"Find all the customers who ordered a bicycle but did not order a helmet." (NOT EXISTS)

"What products have never been ordered?" (NOT IN)

"List customers who have booked entertainers who play country or country rock." (IN)

"Find the entertainers who played engagements for customers Bonnicksen or Rosales." (= SOME)

"Display agents who haven't booked an entertainer." (NOT IN)

"List the entertainers who played engagements for customers Bonnicksen and Rosales." (EXISTS)

"Display students enrolled in a class on Tuesday." (IN)

"Show me the students who have an average score of 85 or better in Art and who also have an average score of 85 or better in Computer Science." (EXISTS)

"Display students who have never withdrawn from a class." (NOT IN)

"List the subjects taught on Wednesday." (IN)

"Display team captains with a current average higher than all other members on their team." (> ALL)

"Show me tournaments that haven't been played yet." (NOT IN)

"Find the bowlers who had a raw score of 170 or better at both Thunderbird Lanes and Bolero Lanes." (EXISTS)

"List all the bowlers who have a current average that's less than all the other bowlers on the same team." (< ALL)

"Show me the recipes that have beef and garlic." (EXISTS)

"Display all the ingredients for recipes that contain carrots." (IN)

"List the ingredients that are used in some recipe where the measurement amount in the recipe is not the default measurement amount." (<> SOME)

"List ingredients not used in any recipe yet." (NOT IN)

Sample Statements

You now know the mechanics of constructing queries using subqueries and have seen some of the types of requests you can answer with a subquery. Let's take a look at a fairly robust set of samples, all of which use one or more subqueries. These examples come from each of the sample databases, and they illustrate the use of the subqueries to either generate an output column or act as a filter.

We've also included sample result sets that would be returned by these operations and placed them immediately after the SQL syntax line. The name that appears immediately above a result set is the name we gave each query in the sample data on the companion CD you'll find bound into the back of the book. We stored each query in the appropriate sample database (as indicated within the example), and we prefixed the names of the queries relevant to this chapter with "CH11." You can follow the instructions in the Introduction of this book to load the samples onto your computer and try them.

❖ **Note** Remember that all the column names and table names used in these examples are drawn from the sample database structures shown in Appendix B, Schema for the Sample Databases. Because many of these examples use complex JOINs, your database system might choose a different way to solve these queries. For this reason, the first few rows might not exactly match the result you obtain, but the total number of rows should be the same. To simplify the process, we have combined the Translation and Clean Up steps for all the following examples.

Subqueries in Expressions

Sales Orders Database

"List vendors and a count of the products they sell to us."

Translation/ Clean Up

Select vendor name ~~and also~~
(select ~~the~~ count(*) ~~of products~~
from ~~the~~ product vendors ~~table~~
where ~~the~~ product vendor ~~table~~ vendor ID
~~equals~~ = ~~the~~ vendors ~~table~~ vendor ID)
from ~~the~~ vendors ~~table~~

SQL

```
SELECT VendName,
   (SELECT COUNT(*)
   FROM Product_Vendors
   WHERE Product_Vendors.VendorID =
      Vendors.VendorID)
AS VendProductCount
FROM Vendors
```

❖ **Note** We assigned an alias name to the subquery in the SELECT clause so that the output displays a meaningful name. If you don't do that, your database system will generate something like Expr1.

CH11_Vendors_Product_Count (10 rows)

VendName	VendProductCount
Shinoman, Incorporated	3
Viscount	6
Nikoma of America	5
ProFormance	3
Kona, Incorporated	1
Big Sky Mountain Bikes	22
Dog Ear	9
Sun Sports Suppliers	5
Lone Star Bike Supply	30
Armadillo Brand	6

Entertainment Agency Database

"Display all customers and the date of the last booking each made."

Translation/ Clean Up

Select customer first name, customer last name, ~~and also~~
(select ~~the highest~~ MAX(start date)
from ~~the~~ engagements ~~table~~
where ~~the~~ engagements ~~table~~ customer ID
~~equals~~ = ~~the~~ customers ~~table~~ customer ID)
from ~~the~~ customers ~~table~~

SQL

```
SELECT Customers.CustFirstName,
   Customers.CustLastName,
   (SELECT MAX(StartDate)
   FROM Engagements
   WHERE Engagements.CustomerID =
      Customers.CustomerID)
AS LastBooking
FROM Customers
```

CH11_Customers_Last_Booking (15 rows)

CustFirstName	CustLastName	LastBooking
Doris	Hartwig	2008-02-23
Deb	Waldal	2008-02-17
Peter	Brehm	2008-02-26
Dean	McCrae	2008-02-24
Elizabeth	Hallmark	2008-02-19
Matt	Berg	2008-02-23
Liz	Keyser	2008-02-19
Darren	Gehring	
Sarah	Thompson	2008-02-24
<< more rows here >>		

❖ **Note** The LastBooking column for some customers is blank (Null) because those customers have no bookings.

School Scheduling Database

"Display all subjects and the count of classes for each subject on Monday."

Translation/ Clean Up

Select subject name ~~and also~~
(select ~~the~~ count(*) ~~of classes~~
from ~~the~~ classes ~~table~~
where Monday schedule ~~is~~ = true
and ~~the~~ classes ~~table~~ subject ID ~~equals~~ = ~~the~~ subjects ~~table~~ subject ID)
from ~~the~~ subjects ~~table~~

SQL

```
SELECT Subjects.SubjectName,
   (SELECT COUNT(*)
   FROM Classes
   WHERE MondaySchedule = 1
   AND Classes.SubjectID = Subjects.SubjectID)
AS MondayCount
FROM Subjects
```

❖ **Note** Be sure to use the test for true that your database system supports. Remember that some database systems require you to compare to a keyword TRUE or to the integer value –1.

CH11_Subjects_Monday_Count (56 rows)

SubjectName	MondayCount
Financial Accounting Fundamentals I	2
Financial Accounting Fundamentals II	1
Fundamentals of Managerial Accounting	1
Intermediate Accounting	1
Business Tax Accounting	1
Introduction to Business	0
Developing A Feasibility Plan	0
Introduction to Entrepreneurship	1
<< more rows here >>	

❖ **Note** Rather than return a Null value when there are no rows, the COUNT aggregate function returns a zero.

Bowling League Database

"Display the bowlers and the highest game each bowled."

Translation/ Clean Up

Select bowler first name, bowler last name~~, and also~~
(select ~~the highest~~ MAX(raw score)
from ~~the~~ bowler scores ~~table~~
where ~~the~~ bowler scores ~~table~~ bowler ID
~~equals~~ = ~~the~~ bowlers ~~table~~ bowler ID)
from ~~the~~ bowlers ~~table~~

SQL

```
SELECT Bowlers.BowlerFirstName,
   Bowlers.BowlerLastName,
   (SELECT MAX(RawScore)
   FROM Bowler_Scores
   WHERE Bowler_Scores.BowlerID =
   Bowlers.BowlerID)
AS HighScore
FROM Bowlers
```

CH11_Bowler_High_Score (32 rows)

BowlerFirstName	BowlerLastName	HighScore
Barbara	Fournier	164
David	Fournier	178
John	Kennedy	191
Sara	Sheskey	149
Ann	Patterson	165
Neil	Patterson	179
David	Viescas	195
Stephanie	Viescas	150
<< more rows here >>		

Recipes Database

"List all the meats and the count of recipes each appears in."

Translation/ Clean Up

Select ingredient class description, ingredient name, ~~and also~~
(select ~~the~~ count(*) ~~of rows~~
from ~~the~~ recipe ingredients ~~table~~
where ~~the~~ recipe ingredients ~~table~~ ingredient ID ~~equals~~ =

~~the~~ ingredients ~~table~~ ingredient ID)
from ~~the~~ ingredient classes ~~table~~
inner join~~ed with the~~ ingredients ~~table~~
on ingredient_classes.ingredient class ID
~~in the ingredients classes table~~
~~matches~~ = ingredients.ingredient class ID ~~in the ingredients table~~
where ingredient class description ~~is~~ = 'meat'

SQL

```
SELECT Ingredient_Classes.IngredientClassDescription,
   Ingredients.IngredientName,
   (SELECT COUNT(*)
   FROM Recipe_Ingredients
   WHERE Recipe_Ingredients.IngredientID =
      Ingredients.IngredientID)
AS RecipeCount
FROM Ingredient_Classes
INNER JOIN Ingredients
ON Ingredient_Classes.IngredientClassID =
   Ingredients.IngredientClassID
WHERE
   Ingredient_Classes.IngredientClassDescription
   = 'Meat'
```

CH11_Meat_Ingredient_Recipe_Count (11 rows)

IngredientClassDescription	IngredientName	RecipeCount
Meat	Beef	2
Meat	Chicken, Fryer	0
Meat	Bacon	0
Meat	Chicken, Pre-cut	0
Meat	T-bone Steak	0
Meat	Chicken Breast	0
Meat	Chicken Leg	1
Meat	Chicken Wing	0
Meat	Chicken Thigh	1
Meat	New York Steak	0
Meat	Ground Pork	1

Subqueries in Filters

Sales Orders Database

"Display customers who ordered clothing or accessories."

Translation/ Clean Up

Select customer ID, customer first name, customer last name
from ~~the~~ customers ~~table~~
where customer ID ~~is equal to~~ = any ~~of the~~
(select~~ion of~~ customer ID
from ~~the~~ orders ~~table~~
inner join~~ed with the~~ order details ~~table~~
on orders.order number ~~in the orders table matches~~
= order_details.order number ~~in the order details table, then~~
inner join~~ed with the~~ products ~~table~~
on products.product number ~~in the products table matches~~
= order_details.product number ~~in the order details table, and then~~ inner join~~ed with the~~ categories ~~table~~
on categories.category ID ~~in the categories table matches~~
= products.category ID ~~in the products table~~
where category description ~~is~~ = 'clothing'
or category description ~~is~~ = 'accessories')

SQL

```
SELECT Customers.CustomerID,
   Customers.CustFirstName,
   Customers.CustLastName
FROM Customers
WHERE Customers.CustomerID = ANY
   (SELECT Orders.CustomerID
   FROM ((Orders
   INNER JOIN Order_Details
   ON Orders.OrderNumber =
      Order_Details.OrderNumber)
   INNER JOIN Products
   ON Products.ProductNumber =
      Order_Details.ProductNumber)
   INNER JOIN Categories
   ON Categories.CategoryID =
      Products.CategoryID
   WHERE Categories.CategoryDescription
      = 'Clothing'
   OR Categories.CategoryDescription
      = 'Accessories')
```

CH11_Customers_Clothing_OR_Accessories (27 rows)

CustomerID	CustFirstName	CustLastName
1001	Suzanne	Viescas
1002	William	Thompson
1003	Gary	Hallmark
1004	Robert	Brown
1005	Dean	McCrae
1006	John	Viescas
1007	Mariya	Sergienko
1008	Neil	Patterson
<< more rows here >>		

❖ **Note** Just for fun, we solved this query by using = ANY. Can you think of a solution using IN or EXISTS? You can find these solutions in the sample database saved as CH11_Customers_Clothing_OR_Accessories_IN and CH11_Customers_Clothing_OR_Accessories_EXISTS.

Entertainment Agency Database

"List the entertainers who played engagements for customers Berg and Hallmark."

❖ **Note** We solved this problem in Chapter 8 with a JOIN of two complex table subqueries. This time, we'll use EXISTS.

Translation/ Clean Up

Select entertainer ID, ~~and~~ entertainer stage name from ~~the~~ entertainers ~~table~~
where ~~there~~ exists
(select * ~~some row~~
from ~~the~~ customers ~~table~~
inner join~~ed with the~~ engagements ~~table~~
on customers.customer ID ~~in the customers table matches~~ = engagements.customer ID ~~in the engagements table~~
where customer last name ~~is~~ = 'Berg'
and ~~the~~ engagements ~~table~~ entertainer ID ~~equals~~ = ~~the~~ entertainers ~~table~~ entertainer ID)~~;~~
and ~~there also~~ exists
(select * ~~some row~~
from ~~the~~ customers ~~table~~
inner join~~ed with the~~ engagements ~~table~~
on customers.customer ID ~~in the customers table matches~~ = engagements.customer ID ~~in the engagements table~~
where customer last name ~~is~~ = 'Hallmark'
and ~~the~~ engagements ~~table~~ entertainer ID ~~equals~~ = ~~the~~ entertainers ~~table~~ entertainer ID)

SQL

```
SELECT Entertainers.EntertainerID,
   Entertainers.EntStageName
FROM Entertainers
WHERE EXISTS
   (SELECT *
   FROM Customers
   INNER JOIN Engagements
   ON Customers.CustomerID =
      Engagements.CustomerID
   WHERE Customers.CustLastName = 'Berg'
   AND Engagements.EntertainerID =
      Entertainers.EntertainerID)
AND EXISTS
   (SELECT *
   FROM Customers
   INNER JOIN Engagements
   ON Customers.CustomerID =
      Engagements.CustomerID
   WHERE Customers.CustLastName = 'Hallmark'
   AND Engagements.EntertainerID =
      Entertainers.EntertainerID)
```

CH11_Entertainers_Berg_AND_Hallmark_EXISTS (4 rows)

EntertainerID	EntStageName
1001	Carol Peacock Trio
1003	JV & the Deep Six
1006	Modern Dance
1008	Country Feeling

School Scheduling Database

"Display students who have never withdrawn from a class."

Translation/ Clean Up

Select student ID, student first name, ~~and~~ student last name
from ~~the~~ students ~~table~~
where ~~the~~ student ID ~~is~~ not in ~~the~~
(select~~ion of~~ student ID
from ~~the~~ student schedules ~~table~~
inner join~~ed with the~~ student class status ~~table~~
on student_schedules.class status
~~in the student schedules table matches~~
= student_class_status.class status
~~in the student class status table~~
where class status description ~~is~~ = 'withdrew')

SQL

```
SELECT Students.StudentID,
   Students.StudFirstName,
   Students.StudLastName
FROM Students
WHERE Students.StudentID NOT IN
   (SELECT Student_Schedules.StudentID
   FROM Student_Schedules
   INNER JOIN Student_Class_Status
   ON Student_Schedules.ClassStatus =
      Student_Class_Status.ClassStatus
   WHERE
   Student_Class_Status.ClassStatusDescription
      = 'Withdrew')
```

❖ **Note** This is a pretty simple query that finds all the students who ever withdrew from a class in the subquery and then asks for all the students NOT IN this list. Can you think how you would solve this with an OUTER JOIN?

CH11_Students_Never_Withdrawn (15 rows)

StudentID	StudFirstName	StudLastName
1002	David	Hamilton
1003	Betsey	Stadick
1004	Janice	Galvin
1005	Doris	Hartwig
1006	Scott	Bishop
1007	Elizabeth	Hallmark
1008	Sara	Sheskey
1010	Marianne	Wier
<< more rows here >>		

Bowling League Database

"Display team captains with a handicap score higher than all other members on their team."

Translation/ Clean Up

Select team name, bowler ID, bowler first name,
bowler last name, ~~and~~ handicap score
from ~~the~~ bowlers ~~table~~
inner join~~ed with the~~ teams ~~table~~
on bowlers.bowler ID ~~in the bowlers table matches~~
= teams.captain ID ~~in the teams table~~
inner join~~ed with the~~ bowler scores ~~table~~
on bowlers.bowler ID ~~in the bowlers table matches~~
= bowler_scores.bowler ID ~~in the bowler scores table~~
where ~~the~~ handicap score ~~is greater than~~ > all ~~the~~
(select~~ion of~~ handicap score
from bowlers as B2
inner join~~ed with the~~ bowler scores ~~table~~ as BS2
on B2.bowler ID ~~in the B2 table matches~~
= BS2.bowler ID ~~in the BS2 table~~
where ~~the~~ B2 ~~table~~ bowler ID ~~is not equal~~ <> ~~the~~ bowlers ~~table~~ bowler ID
and ~~the~~ B2 ~~table~~ team ID ~~is equal~~ = ~~to the~~ bowlers ~~table~~ team ID)

SQL

```
SELECT Teams.TeamName, Bowlers.BowlerID,
   Bowlers.BowlerFirstName,
   Bowlers.BowlerLastName,
   Bowler_Scores.HandiCapScore
FROM (Bowlers
INNER JOIN Teams
ON Bowlers.BowlerID = Teams.CaptainID)
INNER JOIN Bowler_Scores
ON Bowlers.BowlerID = Bowler_Scores.BowlerID
WHERE Bowler_Scores.HandiCapScore > All
  (SELECT BS2.HandiCapScore
   FROM Bowlers AS B2
   INNER JOIN Bowler_Scores AS BS2
     ON B2.BowlerID = BS2.BowlerID
   WHERE B2.BowlerID <> Bowlers.BowlerID
     AND B2.TeamID = Bowlers.TeamID)
```

❖ **Note** We explicitly gave aliases to the second copy of the Bowlers table and the second copy of the Bowler_Scores table in the subquery to make it crystal clear what's going on. We specifically do not want to compare against the score of the current bowler—that would cause the > ALL predicate to fail. We also want to compare only with the other bowlers on the same team.

CH11_Team_Captains_High_Score (1 row)

TeamName	BowlerID	BowlerFirstName	BowlerLastName	HandiCapScore
Huckleberrys	7	David	Viescas	224

Recipes Database

"Display all the ingredients for recipes that contain carrots."

> ❖ **Note** We promised in Chapter 8 that we would show you how to solve this problem with a subquery. We keep our promises!

Translation/ Clean Up

Select recipe title ~~and~~ ingredient name
from ~~the~~ recipes ~~table~~
inner join~~ed with the~~ recipe ingredients ~~table~~
on recipes.recipe ID ~~in the recipes table matches~~
= recipe_ingredients.recipe ID ~~in the recipe ingredients table,~~
~~and then~~ inner join~~ed with the~~ ingredients ~~table~~
on ingredients.ingredient ID ~~in the ingredients table matches~~
= recipe_ingredients.ingredient ID
~~in the recipe ingredients table~~
where recipe ID ~~is~~ in ~~the~~
(select~~ion of~~ recipe ID
from ~~the~~ ingredients ~~table~~
inner join~~ed with the~~ recipe ingredients ~~table~~
on ingredients.ingredient ID ~~in the ingredients table matches~~
= recipe_ingredients.ingredient ID
~~in the recipe ingredients table~~
where ingredient name ~~is~~ = 'carrot')

SQL

```
SELECT Recipes.RecipeTitle,
   Ingredients.IngredientName
FROM (Recipes
INNER JOIN Recipe_Ingredients
ON Recipes.RecipeID =
   Recipe_Ingredients.RecipeID)
INNER JOIN Ingredients
ON Ingredients.IngredientID =
   Recipe_Ingredients.IngredientID
WHERE Recipes.RecipeID
IN
   (SELECT Recipe_Ingredients.RecipeID
   FROM Ingredients
   INNER JOIN Recipe_Ingredients
   ON Ingredients.IngredientID =
      Recipe_Ingredients.IngredientID
   WHERE Ingredients.IngredientName = 'carrot')
```

❖ **Note** If you place the filter for 'carrot' in the outer query, you will see only carrot ingredients in the output. In this problem, we want to see *all* the ingredients from any recipe that uses carrots, so the subquery is a good way to solve it. This query result appears to be sorted by recipe title even though there is no ORDER BY clause. If you want to ensure this sequence in any database system, be sure to include an ORDER BY clause.

CH11_Recipes_Ingredients_With_Carrots (16 rows)

RecipeTitle	IngredientName
Irish Stew	Beef
Irish Stew	Onion
Irish Stew	Potato
Irish Stew	Carrot
Irish Stew	Water
Irish Stew	Guinness Beer
Salmon Filets in Parchment Paper	Salmon
Salmon Filets in Parchment Paper	Carrot
Salmon Filets in Parchment Paper	Leek
<< more rows here >>	

SUMMARY

We began the chapter with a definition of the three types of subqueries defined by the SQL Standard—row, table, and scalar—and recalled that we had already covered how to use table subqueries in a FROM clause. We also briefly described the use of a row subquery and explained that not many commercial implementations support this yet.

Next, we showed how to use a subquery to generate a column expression in a SELECT clause. We discussed a simple example and then introduced two aggregate functions that are useful for fetching related summary information

from another table. (We'll cover all the aggregate functions in detail in the next chapter.)

We then discussed using subqueries to create complex filters in the WHERE clause. We first covered simple comparisons and then introduced special comparison keywords—IN, SOME, ANY, ALL, and EXISTS—that are useful for building predicates with subqueries.

We summarized why subqueries are useful and provided a sample list of problems to solve using subqueries. The rest of the chapter showed examples of how to use subqueries. We broke these examples into two groups: using subqueries in column expressions and using subqueries in filters.

The following section presents a number of requests that you can work out on your own.

Problems for You to Solve

Below, we show you the request statement and the name of the solution query in the sample databases. If you want some practice, you can work out the SQL you need for each request and then check your answer with the query we saved in the samples. Don't worry if your syntax doesn't exactly match the syntax of the queries we saved—as long as your result set is the same.

Sales Orders Database

1. *"Display products and the latest date each product was ordered."*
   (Hint: Use the MAX aggregate function.)
   You can find the solution in CH11_Products_Last_Date (40 rows).
2. *"List customers who ordered bikes."*
   (Hint: Build a filter using IN.)
   You can find the solution in CH11_Customers_Order_Bikes (23 rows).
3. *"Find all customers who ordered a bicycle but did not order a helmet."*
   (Hint: Start with the query above and add a filter using NOT EXISTS.)
   You can find the solution in CH11_Customer_Bikes_No_Helmets (2 rows).
4. *"What products have never been ordered?"*
   (Hint: Build a filter using NOT IN.)
   You can find the solution in CH11_Products_Not_Ordered (2 rows).

Entertainment Agency Database

1. *"Show me all entertainers and the count of each entertainer's engagements."*
   (Hint: Use the COUNT aggregate function.)
   You can find the solution in CH11_Entertainers_Engagement_Count (13 rows).
2. *"List customers who have booked entertainers who play country or country rock."*
   (Hint: Build a filter using IN.)
   You can find the solution in CH11_Customers_Who_Like_Country (13 rows).
3. *"Find the entertainers who played engagements for customers Berg or Hallmark."*
   (Hint: Build a filter using = SOME.)
   You can find the solution in CH11_Entertainers_Berg_OR_Hallmark_SOME (8 rows).
4. *"Display agents who haven't booked an entertainer."*
   (Hint: Build a filter using NOT IN.)
   You can find the solution in CH11_Bad_Agents (1 row).

School Scheduling Database

1. *"List all staff members and the count of classes each teaches."*
   (Hint: Use the COUNT aggregate function.)
   You can find the solution in CH11_Staff_Class_Count (27 rows).
2. *"Display students enrolled in a class on Tuesday."*
   (Hint: Build a filter using IN.)
   You can find the solution in CH11_Students_In_Class_Tuesdays (18 rows).
3. *"Show me the students who have an average score of 85 or better in Art and who also have an average score of 85 or better in Computer Science."*
   (Hint: Build a filter using EXISTS.)
   You can find the solution in CH11_Good_Art_CS_Students_EXISTS (1 row).
4. *"List the subjects taught on Wednesday."*
   (Hint: Build a filter using IN.)
   You can find the solution in CH11_Subjects_On_Wednesday (45 rows).

Bowling League Database

1. *"Show me all the bowlers and a count of games each bowled."*
   (Hint: Use the COUNT aggregate function.)
   You can find the solution in CH11_Bowlers_And_Count_Games (32 rows).

2. *"Show me tournaments that haven't been played yet."*
   (Hint: Use a NOT IN filter.)
   You can find the solution in CH11_Tourneys_Not_Played (6 rows).
3. *"Find the bowlers who had a raw score of 170 or better at both Thunderbird Lanes and Bolero Lanes."*
   (Hint: Build a filter using EXISTS.)
   You can find the solution in CH11_Good_Bowlers_TBird_And_Bolero_EXISTS (11 rows).
4. *"List all the bowlers who have a raw score that's less than all of the other bowlers on the same team."*
   (Hint: Build a filter using < ALL. Also use DISTINCT in case a bowler has multiple games with the same low score.)
   You can find the solution in CH11_Bowlers_Low_Score (3 rows).

Recipes Database

1. *"Show me the types of recipes and the count of recipes in each type."*
   (Hint: Use the COUNT aggregate function.)
   You can find the solution in CH11_Count_Of_Recipe_Types (7 rows).
2. *"Show me the recipes that have beef and garlic."*
   (Hint: Build a filter using EXISTS.)
   You can find the solution in CH11_Recipes_Beef_And_Garlic (1 row).
3. *"List the ingredients that are used in some recipe where the measurement amount in the recipe is not the default measurement amount."*
   (Hint: Build a filter using <> SOME.)
   You can find the solution in CH11_Ingredients_Using_NonStandard_Measure (21 rows).
4. *"List ingredients not used in any recipe yet."*
   (Hint: Build a filter using NOT IN.)
   You can find the solution in CH11_Ingredients_No_Recipe (20 rows).

Part IV
Summarizing and Grouping Data

This page intentionally left blank

12

Simple Totals

"There are two kinds of statistics: the kind you look up and the kind you make up."
—Rex Stout
Death of a Doxy: A Nero Wolfe Novel

Topics Covered in This Chapter

Aggregate Functions

Using Aggregate Functions in Filters

Sample Statements

Summary

Problems for You to Solve

You now know how to select the columns you need for a given request, define expressions that add extra levels of detail, join the appropriate tables that supply the columns you require, and define conditions to filter the data sent to the result set. We've shown you all these techniques so that you can learn how to retrieve detailed information from one or more tables in the database. In this and the next two chapters, we'll show you how to take a step back and look at the data from a much broader perspective, otherwise known as "seeing the big picture."

In this chapter, you'll learn how to use aggregate functions to produce basic summary information. In Chapter 13, Grouping Data, we'll show you how to organize data into groups with the GROUP BY clause of the SELECT statement, and in Chapter 14, Filtering Grouped Data, we'll show you various filtering techniques you can apply to the data after it is grouped.

Aggregate Functions

The requests you've been working with so far have required answers involving individual column values from the rows returned by the FROM and WHERE clauses. However, you'll often encounter requests, such as the following, that require only calculated values across multiple rows for an answer.

"How many of our customers live in Seattle?"

"What is the lowest price and highest price we've assigned to any item in our inventory?"

"How many classes is Mike Hernandez teaching?"

"What time does our earliest class begin?"

"What is the average length of a class?"

"What is the total amount for order number 12?"

The SQL Standard provides a set of *aggregate functions* that allow you to calculate a single value from the rows in a result set or from the values returned by a value expression. You can apply a given function to all the rows or values, or you can use a WHERE clause to apply the function to a specific set of rows or values. For example, you can use an aggregate function to determine the largest or smallest value of a value expression, count the number of rows in a result set, or calculate a total using only distinct values from a value expression. Figure 12–1 shows the syntax for the basic aggregate functions supported by all database systems.

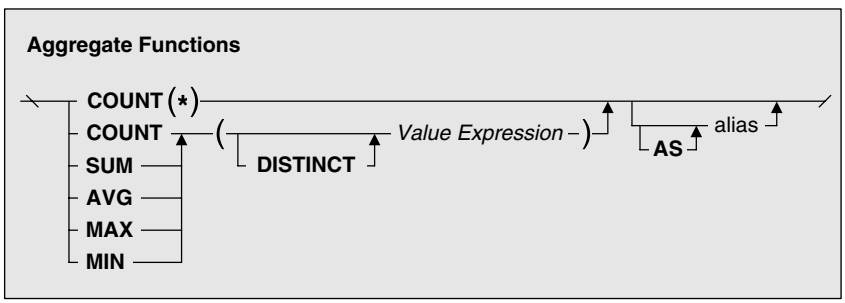


Figure 12–1 *The syntax diagram for aggregate functions*

As you can see, aggregate functions have a very simple and straightforward syntax. In the previous chapter, we discussed using two of the aggregate functions in a subquery either to return a single calculated value in a SELECT

clause or to fetch a calculated value that you can use in a predicate in a WHERE clause. We'll show you a few more examples of this usage in this chapter.

❖ **Note** The 2003 SQL Standard defines a dozen or more additional aggregate operations, but many are not yet implemented in any major commercial database system. In this chapter, we focus on the basic aggregate functions supported by all major systems. After you learn how to work with these, consult your database documentation to learn whether more functions are available to use in your SQL statements.

Each aggregate function returns a single value, regardless of whether it is processing the rows in a result set or the values returned by a value expression. With the exception of COUNT(*), all aggregate functions automatically disregard Null values. You can use several aggregate functions at the same time in the list of value expressions immediately following the SELECT keyword, and you can even mix value expressions containing aggregate functions with value expressions containing literal values. But you need to be careful once you've started including aggregate expressions.

When you include an aggregate expression, you're asking your database system to calculate one value across a group of rows. You'll learn in the next chapter that you can define the groups you want by using the GROUP BY clause. However, in this chapter, we're looking at simple queries that do not explicitly specify groups. In the absence of a group specification, the group of records that your database uses to calculate any aggregate expression is all the rows returned by your FROM and WHERE clauses.

If you think about it, it doesn't make sense to also include a value expression using a column from one of your tables that isn't inside an aggregate function. Remember that we introduced you to the COUNT and MAX aggregate functions in Chapter 11, Subqueries. Consider the following SQL.

```
SQL         SELECT LastName, COUNT(*) As CountOfStudents
            FROM Students
```

Including the COUNT function without specifying any groups asks your database system to count all the rows in the result set returned from the FROM clause. COUNT(*) returns a single value—the count of all the rows in the

Students table—so the query should return one row. Which LastName should your database system display? The answer is it can't figure out which one to choose, so the above statement is illegal.

It is valid, however, to include a literal expression to further enhance your output. You can do this because a literal expression is simply a constant—it has the same value for all rows. So, it's perfectly legal to use the following SQL.

```
SQL         SELECT 'The number of students is: ', COUNT(*)
               As CountOfStudents
            FROM Students
```

This returns one row:

The number of students is:	18

Now that we've gotten that little warning out of the way, let's look at each of these aggregate functions and how you might use them to answer a request.

Counting Rows and Values with COUNT

The SQL Standard defines two versions of the COUNT function. COUNT(*) processes rows in a result set, and COUNT (*value expression*) processes values returned by a value expression.

Counting All the Rows

You use COUNT(*) to determine how many rows exist in a result set. The COUNT(*) function counts *all* the rows in a result set, including redundant rows and rows containing Null values. Here's a simple example of the type of question you can answer with this function.

> ❖ **Note** Throughout this chapter, we use the "Request/Translation/Clean Up/SQL" technique introduced in Chapter 4, Creating a Simple Query. All examples assume you have thoroughly studied and understood the concepts covered in previous chapters, especially the chapters on JOINs and subqueries.

"Show me the total number of employees we have in our company."

Translation Select the count of employees from the employees table

Clean Up Select ~~the~~ count ~~of employees~~ (*)
from ~~the~~ employees ~~table~~

```
SQL         SELECT COUNT(*)
            FROM Employees
```

Note that we use "(*)" in the Clean Up statement to indicate that we want to count *all* the rows in the Employees table. You should add the asterisk in your Clean Up step when you work with this type of request because it helps ensure that you use the correct COUNT function. The SELECT statement in this example generates a result set consisting of a single-column row containing a numeric value that represents the total number of rows in the Employees table.

There is no restriction on the number of rows the COUNT(*) function processes. You can indicate which rows COUNT(*) should include by using a WHERE clause. For example, here's how you define a SELECT statement that counts all the rows in the Employees table for those employees who live in Washington state.

```
SQL         SELECT COUNT(*)
            FROM Employees
            WHERE EmpState = 'WA'
```

As we work through this chapter, you'll see that you can use a WHERE clause to filter the rows or values processed by any aggregate function.

When you use an aggregate function in a SELECT statement, you might or might not see a column name in the result set for the return value of the function. Some database systems provide a default column name, and others do not. But you can use the AS option of the function's syntax to provide a meaningful column name for the result set. Here's how you might apply this option to the previous example.

```
SQL         SELECT COUNT(*) AS TotalWashingtonEmployees
            FROM Employees
            WHERE EmpState = 'WA'
```

Now the result set consists of a column called TotalWashingtonEmployees that contains the return value of the COUNT(*) function. As the syntax

diagram in Figure 12–1 indicates, you can apply this technique to any aggregate function.

Counting Values in a Column or Expression

You use the COUNT(*value expression*) function to count the total number of *non-Null* values returned by a value expression. (This expression is more commonly known as COUNT, which is the name we'll use for the remainder of the book.) It counts all values returned by a value expression, regardless of whether they are unique or duplicate, but automatically excludes any Null values from the final count. You can use COUNT to answer this type of request.

"How many customers were able to indicate which county they live in?"

Here you need to determine how many actual values exist in the county column. Remember that COUNT(*) *includes* Null values as well, so it won't provide you with the correct answer. Instead, you use the COUNT function and translate the request in this manner.

Translation	Select the count of non-Null county values as NumberOfKnownCounties from the customers table
Clean Up	Select ~~the~~ count ~~of non-Null~~ (county) ~~values~~ as NumberOfKnownCounties from ~~the~~ customers ~~table~~

SQL	`SELECT COUNT(CustCounty)`
	`  AS NumberOfKnownCounties`
	`FROM Customers`

Note that the Translation and Clean Up statements explicitly ask for non-Null values. Although you already know that this function processes only non-Null values, it's a good idea to add this to both statements so that you'll be sure to use the correct COUNT function. The SELECT statement defined here will generate a single row that contains a numeric value representing the count of rows containing non-Null county names found in the CustCounty column.

Remember that the COUNT function treats duplicate county names as though they were unique and includes every one of them in the final count. You can, however, use the function's DISTINCT option to exclude duplicate values from the count. The next example shows how you might apply it to a given request.

"How many unique county names are there in the customers table?"

Translation: Select the count of unique non-Null county names as NumberOfUniqueCounties from the customers table

Clean Up: Select ~~the~~ count ~~of unique non-Null~~ (distinct county) ~~names~~ as NumberOfUniqueCounties from ~~the~~ customers ~~table~~

SQL:
```
SELECT COUNT(DISTINCT CustCounty)
   AS NumberOfUniqueCounties
FROM Customers
```

When you use the DISTINCT option, the database retrieves all the non-Null values from the county column, eliminates the duplicates, and *then* counts the values that remain. The database goes through much of this same process whenever you use DISTINCT with the SUM, AVG, MIN, or MAX functions.

In this next example, we use a slightly altered version of the previous request to show that you can apply a filter to the COUNT function.

"How many unique county names are there in the customers table for the state of Oregon?"

Translation: Select the count of unique non-Null county names as NumberOfUniqueOregonCounties from the customers table where the state is 'OR'

Clean Up: Select ~~the~~ count ~~of unique non-Null~~ (distinct county) ~~names~~ as NumberOfUniqueOregonCounties from ~~the~~ customers ~~table~~ where ~~the~~ state ~~is~~ = 'OR'

SQL:
```
SELECT COUNT(DISTINCT CustCounty)
   AS NumberOfUniqueOregonCounties
FROM Customers
WHERE CustState = 'OR'
```

It's important to note that you *cannot* use DISTINCT with COUNT(*). This is a reasonable restriction because COUNT(*) counts *all* rows in a table, regardless of whether any are redundant or contain Null values.

Computing a Total with SUM

You can calculate a total for a *numeric* value expression with the SUM function. It processes all the non-Null values of the value expression and returns a final total to the result set. Note that if the value expression in all the rows is Null or

if the result of evaluating the FROM and WHERE clauses is an empty set, then SUM returns a Null. Here's a sample request you can answer with SUM.

"What is the total amount we pay in salaries to our employees in California?"

Translation	Select the sum of salary as TotalSalaryAmount from the employees table where the state is 'CA'
Clean Up	Select ~~the~~ sum ~~of~~ (salary) as TotalSalaryAmount from ~~the~~ employees ~~table~~ where ~~the~~ state ~~is~~ = 'CA'

SQL	`SELECT SUM(Salary) AS TotalSalaryAmount`
	`FROM Employees`
	`WHERE EmpState = 'CA'`

The value expression we used here was a simple column reference. However, you can also use SUM on a value expression consisting of a numeric expression, as we demonstrate in the next example.

"How much is our current inventory worth?"

Translation	Select the sum of wholesale price times quantity on hand as TotalInventoryValue from the products table
Clean Up	Select ~~the~~ sum ~~of~~ (wholesale price ~~times~~ * quantity on hand) as TotalInventoryValue from ~~the~~ products ~~table~~

SQL	`SELECT SUM(WholesalePrice * QuantityOnHand)`
	`  AS TotalInventoryValue`
	`FROM Products`

As you know, a row must contain actual values in the WholesalePrice and QuantityOnHand columns in order for it to be processed by the SUM function. In this instance, the database processes the expression for all qualifying rows in the Products table, totals the results with the SUM function, and then sends the grand total to the result set.

Here's an example of how to use SUM to calculate a total for a unique set of numeric values.

"Calculate a total of all unique wholesale costs for the products we sell."

Translation	Select the sum of unique wholesale costs as SumOfUniqueWholesaleCosts from the products table

Clean Up
Select ~~the~~ sum ~~of unique~~ (distinct wholesale costs)
as SumOfUniqueWholesaleCosts
from ~~the~~ products ~~table~~

SQL
```
SELECT SUM(DISTINCT WholesaleCost)
   AS SumOfUniqueWholesaleCosts
FROM Products
```

Calculating a Mean Value with AVG

Another function you can use on *numeric* values is AVG, which calculates the arithmetic mean of all non-Null values returned by a value expression. You can use AVG to answer a request such as this.

"What is the average contract amount for vendor number 10014?"

Translation
Select the average of contract price as AverageContractPrice
from the vendor contracts table where the vendor ID is
10014

Clean Up
Select ~~the average of~~ avg (contract price)
as AverageContractPrice
from ~~the~~ vendor contracts ~~table~~
where the vendor ID ~~is~~ = 10014

SQL
```
SELECT AVG(ContractPrice)
   AS AverageContractPrice
FROM Vendor_Contracts
WHERE VendorID = 10014
```

As you work with your Clean Up statement, be sure to cross out the word "average" and replace it with "avg" to help keep you from accidentally using "Average" in the SELECT clause. "Average" is not a valid SQL keyword, so the SELECT statement will fail if you try to use it.

You can also use AVG to process a numeric expression, just as you did with the SUM function. Remember that you cannot use AVG with a value expression that is not numeric. Most database systems will give you an error if you try to use these functions with character string or datetime data.

"What is the average item total for order 64?"

Translation
Select the average of price times quantity ordered as
AverageItemTotal from the order details table where order ID
is 64

Clean Up: Select ~~the average of~~ avg (price ~~times~~ * quantity ordered)
as AverageItemTotal
from ~~the~~ order details ~~table~~
where order ID ~~is~~ = 64

SQL:
```
SELECT AVG(Price * QuantityOrdered)
   AS AverageItemTotal
FROM Order_Details
WHERE OrderID = 64
```

Keep in mind that a row must contain actual values in the columns Price and QuantityOrdered in order for that row to be processed by the AVG function. Otherwise, the numeric expression evaluates to Null, and the AVG function disregards the row entirely. As with SUM, if the value expression in all rows is Null or the result of evaluating the FROM and WHERE clauses is an empty set, AVG returns a Null value.

In this next example, we use the DISTINCT option to average a unique set of numeric values.

"Calculate an average of all unique product prices."

Translation: Select the average of unique prices as UniqueProductPrices
from the products table

Clean Up: Select ~~the average of unique~~ avg (distinct price~~s~~)
as UniqueProductPrices
from ~~the~~ products ~~table~~

SQL:
```
SELECT AVG(DISTINCT Price)
   AS UniqueProductPrices
FROM Products
```

Finding the Largest Value with MAX

You can determine the *largest* value returned by a value expression with the MAX function. The MAX function can process any type of data, and the value it returns depends on the data it processes.

CHARACTER STRINGS: The value that MAX returns is based on the collating sequence used by your database system or computer. For example, if your database uses the ASCII character set and is case insensitive, it sorts company names in this manner: ". . . 4th Dimension Productions . . . Al's Auto Shop . . . allegheny & associates . . . Zercon Productions . . . zorn credit services." In this instance, MAX will return "zorn credit services" as the MAX value.

NUMBERS	MAX returns the largest number.
DATETIME	MAX evaluates dates and times in chronological order and returns the *most recent* (or latest) date or time.

Here are a couple of examples of how you might use MAX to answer a request.

"What is the largest amount we've paid on a contract?"

Translation	Select the maximum contract price as LargestContractPrice from the engagements table
Clean Up	Select ~~the~~ max~~imum~~ (contract price) as LargestContractPrice from ~~the~~ engagements ~~table~~

SQL	`SELECT MAX(ContractPrice)`
	`  AS LargestContractPrice`
	`FROM Engagements`

"What was the largest line item total for order 3314?"

Translation	Select the maximum price times quantity ordered as LargestItemTotal from the order details table where the order ID is 3314
Clean Up	Select ~~the~~ max~~imum~~ (price ~~times~~ * quantity ordered) as LargestItemTotal from ~~the~~ order details ~~table~~ where ~~the~~ order ID ~~is~~ = 3314

SQL	`SELECT MAX(Price * QuantityOrdered)`
	`  AS LargestItemTotal`
	`FROM Order_Details`
	`WHERE OrderID = 3314`

You might be tempted to use the DISTINCT option to return a unique instance of the highest or most recent value. Although the SQL Standard specifies DISTINCT as an option for the MAX function, DISTINCT *has no effect* on the MAX function whatsoever. There can be only one maximum value, regardless of whether or not it is distinct. For example, if you're looking for the most recent hire date in the Agents table, both of the following expressions return the same value.

```
SELECT MAX(DateHired) FROM Agents
SELECT MAX(DISTINCT DateHired) FROM Agents
```

We present both versions of the function because they are part of the current SQL Standard, but we recommend that you use the MAX function without the DISTINCT option. When you include DISTINCT, you're asking your database system to do extra and unnecessary work to first find the unique values and then figure out which one is the largest or latest.

Finding the Smallest Value with MIN

The MIN function allows you to determine the *smallest* value returned by a value expression. It works like the MAX function but returns the opposite value: the first character string (based on the collating sequence), the smallest number, and the earliest date or time.

You can answer requests such as these with the MIN function.

"What is the lowest price we charge for a product?"

Translation	Select the minimum price as LowestProductPrice from the products table
Clean Up	Select ~~the~~ min~~imum~~ (price) as LowestProductPrice from ~~the~~ products ~~table~~
SQL	`SELECT MIN(Price) AS LowestProductPrice FROM Products`

"What was the lowest line item total for order 3314?"

Translation	Select the minimum price times quantity ordered as LowestItemTotal from the order details table where the order ID is 3314
Clean Up	Select ~~the~~ min~~imum~~ (price ~~times~~ * quantity ordered) as LowestItemTotal from ~~the~~ order details ~~table~~ where ~~the~~ order ID ~~is~~ = 3314
SQL	`SELECT MIN(Price * QuantityOrdered) AS LowestItemTotal FROM Order_Details WHERE OrderID = 3314`

It's important to note that the DISTINCT option *has no effect* whatsoever on the MIN function. (As you know, this was the case with the MAX function as well.) There can be only one minimum value, regardless of whether or not it

is distinct. For example, both of the following expressions return the same value.

```
SELECT MIN(DateHired) FROM Agents
SELECT MIN(DISTINCT DateHired) FROM Agents
```

We present both versions of the function because they are part of the current SQL Standard, but we recommend that you use the MIN function without the DISTINCT option. When you include DISTINCT, you're asking your database system to do extra and unnecessary work to first find the unique values and then figure out which one is the lowest or earliest.

Using More Than One Function

As we mentioned at the beginning of this section, you can use several aggregate functions at the same time. This gives you the ability to show contrasting information using a single SELECT statement. For example, you can use the MIN and MAX functions to show the earliest and most recent order dates for a specific customer, or the MAX, MIN, and AVG functions to show the highest, lowest, and average grades for a given student. Here are other examples of how you might use two or more aggregate functions.

"Show me the earliest and most recent review dates for the employees in the advertising department."

Translation: Select the minimum review date as EarliestReviewDate and the maximum review date as RecentReviewDate from the employees table where the department is 'Advertising'

Clean Up:
Select ~~the~~ min~~imum~~ review date as EarliestReviewDate,
~~and the~~ max~~imum~~ review date as RecentReviewDate
from ~~the~~ employees ~~table~~
where ~~the~~ department ~~is~~ = 'Advertising'

SQL:
```
SELECT MIN(ReviewDate) AS EarliestReviewDate,
   MAX(ReviewDate) AS RecentReviewDate
FROM Employees
WHERE Department = 'Advertising'
```

"How many different products were ordered on order number 553, and what was the total cost of that order?"

Translation: Select the count of product ID as TotalProductsPurchased and the sum of price times quantity ordered as OrderAmount from the order details table where the order number is 553

Clean Up	Select ~~the~~ count ~~of~~ (product ID) as TotalProductsPurchased, ~~and the~~ sum ~~of~~ (price ~~times~~ * quantity ordered) as OrderAmount from ~~the~~ order details ~~table~~ where ~~the~~ order number ~~is~~ = 553

SQL	`SELECT COUNT(ProductID) AS TotalProductsPurchased,`
	`  SUM(Price * QuantityOrdered) AS OrderAmount`
	`FROM Order_Details`
	`WHERE OrderNumber = 553`

You must keep in mind a couple of restrictions when you work with two or more aggregate functions. The first is that you cannot embed one aggregate function within another. This restriction makes the following expression illegal.

```
SUM(AVG(LineItemTotal))
```

The second is that you cannot use a subquery as the value expression of an aggregate function. For example, the following expression is illegal under this restriction.

```
AVG((SELECT Price FROM Products WHERE Category = 'Bikes'))
```

Despite these restrictions, you've learned how easily you can use aggregate functions in a SELECT clause to retrieve relatively complex statistical information. Let's now look at how you might use aggregate functions to filter the information in a result set.

Using Aggregate Functions in Filters

Because an aggregate function returns a single value, you can use it as part of a comparison predicate in a search condition. You have to place the aggregate function within a subquery, however, and then use the subquery as part of the comparison predicate. If you're thinking that this sounds familiar, you're right. In Chapter 11, you learned how to use a subquery as part of a search condition in a WHERE clause and an aggregate function within a subquery. So you already know, in a general sense, how to use an aggregate function to filter the data sent to a result set. Now let's expand on that knowledge.

Using an aggregate function as part of a comparison predicate allows you to test the value of a value expression against a single statistical value. Although

you could use a literal value for the task, a subquery gives you more flexibility and provides a more dynamic aspect to the condition. For example, suppose you're making the following request to the database.

> *"List the engagement numbers that have a contract price greater than or equal to the overall average contract price."*

One method you can use to answer this request is to calculate the overall average contract price manually and then plug that specific value into a comparison predicate.

Translation	Select the engagement number from the engagements table where the contract price is greater than or equal to $24,887.00
Clean Up	Select ~~the~~ engagement number from ~~the~~ engagements ~~table~~ where ~~the~~ contract price ~~is greater than or equal to~~ >= ~~$~~24,887.00

SQL	`SELECT EngagementNumber`
	`FROM Engagements`
	`WHERE ContractPrice >= 24887.00`

Hey, why do more work than necessary? You can use an aggregate function in a subquery and let the database system do the work for you.

Translation	Select the engagement number from the engagements table where the contract price is greater than or equal to the overall average contract price in the engagements table
Clean Up	Select ~~the~~ engagement number from ~~the~~ engagements ~~table~~ where ~~the~~ contract price ~~is greater than or equal to the~~ >= ~~overall~~ (select ~~average~~ avg contract price ~~in the~~ from engagements ~~table~~)

SQL	`SELECT EngagementNumber`
	`FROM Engagements`
	`WHERE ContractPrice >=`
	`(SELECT AVG(ContractPrice)`
	`FROM Engagements)`

It should be obvious that using a subquery with an aggregate function is your best course of action. If you use a literal value, you must be certain that you always recalculate the average contract price before executing the SELECT

statement, just in case you've modified any existing contract prices. You then have to make sure that you enter the value correctly in the comparison predicate. But you won't have to worry about any of this if you use a subquery instead. The AVG function is always evaluated whenever you execute the SELECT statement, and it always returns the correct value regardless of whether you've modified any of the contract prices. (This is true for any aggregate function you use in a subquery.)

You can limit the rows that an aggregate function evaluates by using a WHERE clause in the subquery. This allows you to narrow the scope of the statistical value returned by the aggregate function. You already learned how to apply a WHERE clause to a subquery back in Chapter 11, so let's look at an example of how you might apply this technique.

"List the engagement number and contract price of all engagements that have a contract price larger than the total amount of all contract prices for the entire month of September 2007."

Translation: Select engagement number and contract price from the engagements table where the contract price is greater than the sum of all contract prices of engagements dated between September 1, 2007, and September 30, 2007

Clean Up:

```
Select engagement number, ~~and~~ contract price
from ~~the~~ engagements ~~table~~
where ~~the~~ contract price ~~is greater than~~ > ~~the~~
(select sum ~~of all~~ (contract prices)
from engagements
where ~~dated~~ start date between ~~September 1, 2007~~,
'2007-09-01'
and ~~September 30, 2007~~ '2007-09-30')
```

SQL:

```
SELECT EngagementNumber, ContractPrice
FROM Engagements
WHERE ContractPrice >
   (SELECT SUM(ContractPrice) FROM Engagements
      WHERE StartDate BETWEEN '2007-09-01'
      AND '2007-09-30')
```

You might find that you rarely have a need to use aggregate functions in filters, but they certainly come in handy when you have to answer those occasional off-the-wall requests.

Sample Statements

In this chapter, you've learned how to use aggregate functions in a SELECT clause and within a subquery being used as part of a comparison predicate. Now let's look at some examples of working with aggregate functions using the tables from each of the sample databases. These examples illustrate the use of the aggregate functions as output columns and in subqueries.

We've also included sample result sets that would be returned by these operations and placed them immediately after the SQL syntax line. The name that appears immediately above a result set is the name we gave each query in the sample data on the companion CD you'll find bound into the back of the book. We stored each query in the appropriate sample database (as indicated within the example), and we prefixed the names of the queries relevant to this chapter with "CH12." You can follow the instructions in the Introduction of this book to load the samples onto your computer and try them.

❖ **Note** Remember that all the column names and table names used in these examples are drawn from the sample database structures shown in Appendix B, Schema for the Sample Databases. To simplify the process, we have combined the Translation and Clean Up steps for all the following examples.

Sales Orders Database

"How many customers do we have in the state of California?"

Translation/ Clean Up:
Select ~~the~~ count(*) as NumberOfCACustomers ~~of all customers~~ from ~~the~~ customers ~~table~~ where ~~the~~ state ~~is~~ = 'CA'

SQL:

```
SELECT COUNT(*) AS NumberOfCACustomers
FROM Customers
WHERE CustState = 'CA'
```

CH12_Number_Of_California_Customers (1 Row)

NumberOfCACustomers
7

"List the product names and numbers that have a quoted price greater than or equal to the overall average retail price in the products table."

Translation/ Clean Up

Select ~~the~~ product name, ~~and the~~ product number
from ~~the~~ products ~~table~~
inner join~~ed with the~~ order details ~~table~~
on products.product number ~~in the products table matches~~
= order_details.product number ~~in the order details table~~
where ~~the~~ quoted price ~~is greater than or equal to~~ >=
(select ~~the average~~ avg(retail price) ~~in the~~ from products ~~table~~)

SQL

```
SELECT DISTINCT Products.ProductName,
   Products.ProductNumber
FROM Products
INNER JOIN Order_Details
ON Products.ProductNumber =
   Order_Details.ProductNumber
WHERE Order_Details.QuotedPrice >=
   (SELECT AVG(RetailPrice)
   FROM Products)
```

❖ **Note** We chose to ask for DISTINCT products because (we hope) a particular product might have been ordered more than once. We need to see each product name and number only once.

CH12_Quoted_Price_vs_Average_Retail_Price (4 Rows)

ProductName	ProductNumber
Eagle FS-3 Mountain Bike	2
GT RTS-2 Mountain Bike	11
Trek 9000 Mountain Bike	1
Viscount Mountain Bike	6

Entertainment Agency Database

"List the engagement number and contract price of contracts that occur on the earliest date."

Translation/ Clean Up
Select engagement number, ~~and~~ contract price
from ~~the~~ engagements ~~table~~
where ~~the~~ start date ~~is equal to the~~ =
~~earliest~~ (select min(start date) ~~in the~~ from engagements ~~table~~)

SQL

```
SELECT EngagementNumber, ContractPrice
FROM Engagements
WHERE StartDate =
   (SELECT MIN(StartDate) FROM Engagements)
```

CH12_Earliest_Contracts (1 Row)

EngagementNumber	ContractPrice
2	$200.00

"What was the total value of all engagements booked in October 2007?"

Translation/ Clean Up
Select ~~the~~ sum ~~of~~ (contract price) as TotalBookedValue
from ~~the~~ engagements ~~table~~
where ~~the~~ start date ~~is~~ between ~~October 1, 2007~~ '2007-10-01'
and ~~October 31, 2007~~ '2007-10-31'

SQL

```
SELECT SUM(ContractPrice) AS TotalBookedValue
FROM Engagements
WHERE StartDate
   BETWEEN '2007-10-01' AND '2007-10-31'
```

CH12_Total_Booked_Value_For_October_2007 (1 Row)

TotalBookedValue
$27,755.00

School Scheduling Database

"What is the largest salary we pay to any staff member?"

Translation/ Clean Up: Select ~~the~~ max~~imum~~ (salary) as LargestStaffSalary
from ~~the~~ staff ~~table~~

SQL:
```
SELECT Max(Salary) AS LargestStaffSalary
FROM Staff
```

CH12_Largest_Staff_Salary (1 Row)

LargestStaffSalary
$60,000.00

"What is the total salary amount paid to our staff in California?"

Translation/ Clean Up: Select ~~the~~ sum ~~of~~ (salary) as TotalAmountPaid
from ~~the~~ staff ~~table~~
~~for all our California staff~~ where state = 'CA'

SQL:
```
SELECT SUM(Salary) AS TotalAmountPaid
FROM Staff
WHERE StfState = 'CA'
```

CH12_Total_Salary_Paid_To_California_Staff (1 Row)

TotalAmountPaid
$209,000.00

Bowling League Database

"How many tournaments have been played at Red Rooster Lanes?"

Translation/ Clean Up

Select ~~the~~ count ~~of~~ (tourney location)~~s~~ as NumberOfTournaments
from ~~the~~ tournaments ~~table~~
where ~~the~~ tourney location ~~is~~ = 'Red Rooster Lanes'

SQL

```
SELECT COUNT(TourneyLocation)
   AS NumberOfTournaments
FROM Tournaments
WHERE TourneyLocation = 'Red Rooster Lanes'
```

CH12_Number_Of_Tournaments_At_Red_Rooster_Lanes (1 Row)

NumberOfTournaments
3

"List the last name and first name, in alphabetical order, of every bowler whose personal average score is greater than or equal to the overall average score."

Translation/ Clean Up

Select ~~the~~ last name, ~~and~~ first name from ~~the~~ bowlers ~~table~~
where ~~the~~ (select ~~average~~ avg(raw score)
from ~~the~~ bowlers scores ~~table~~ as BS
~~for the current bowler~~ where BS.bowler ID = bowlers.bowler ID)
~~is greater than or equal to the~~ >=
~~overall~~ (select avg(raw score) ~~score in the~~ from bowler scores ~~table~~)
~~sorted~~ order by last name, ~~and~~ first name

SQL

```
SELECT Bowlers.BowlerLastName,
   Bowlers.BowlerFirstName
FROM Bowlers
WHERE (SELECT AVG(RawScore)
FROM Bowler_Scores AS BS
WHERE BS.BowlerID = Bowlers.BowlerID)
>=(SELECT AVG(RawScore) FROM Bowler_Scores)
ORDER BY Bowlers.BowlerLastName,
         Bowlers.BowlerFirstName
```

CH12_Better_Than_Overall_Average (17 Rows)

BowlerLastName	BowlerFirstName
Cunningham	David
Fournier	David
Hallmark	Alaina
Hallmark	Gary
Hernandez	Michael
Kennedy	Angel
Kennedy	John
Patterson	Kathryn
Patterson	Neil
Patterson	Rachel
Pundt	Steve
Thompson	Mary
Thompson	Sarah
Thompson	William
Viescas	Caleb
Viescas	David
Viescas	John

❖ **Note** You can see that this is a creative use of two subqueries in the WHERE clause to solve the problem.

Recipes Database

"How many recipes contain a beef ingredient?"

Translation/ Clean Up

Select ~~the~~ count (*) ~~of recipes~~ as NumberOfRecipes
from ~~the~~ recipes ~~table~~
where ~~the~~ recipe ID ~~is~~ in ~~the~~
(selection ~~of~~ recipe IDs ~~in the~~
from recipe ingredients ~~table~~
inner join~~ed with the~~ ingredients ~~table~~
on recipe_ingredients.ingredient ID
~~in the recipe ingredients table~~
~~matches~~ = ingredients.ingredient ID ~~in the ingredients table~~
where ~~the~~ ingredient name ~~is~~ like 'Beef%')

SQL

```
SELECT COUNT(*) AS NumberOfRecipes
FROM Recipes
WHERE Recipes.RecipeID IN
   (SELECT RecipeID
   FROM Recipe_Ingredients
   INNER JOIN Ingredients ON
      Recipe_Ingredients.IngredientID =
      Ingredients.IngredientID
   WHERE Ingredients.IngredientName
      LIKE 'Beef%')
```

CH12_Recipes_With_Beef_Ingredient (1 Row)

NumberOfRecipes
3

"How many ingredients are measured by the cup?"

Translation/ Clean Up

Select ~~the~~ count (*) ~~of ingredients~~ as NumberOfIngredients
from ~~the~~ ingredients ~~table~~
inner join~~ed with the~~ measurements ~~table~~
on ingredients.measure amount ID
~~in the ingredients table matches~~
= measurements.measure amount ID
~~in the measurements table~~
where ~~the~~ measurement description ~~is~~ = 'Cup'

SQL

```
SELECT COUNT(*) AS NumberOfIngredients
FROM Ingredients
INNER JOIN Measurements
ON Ingredients.MeasureAmountID =
   Measurements.MeasureAmountID
WHERE MeasurementDescription = 'Cup'
```

CH12_Number_of_Ingredients_ Measured_by_the_Cup (1 Row)

NumberOfIngredients
12

SUMMARY

We began this chapter by introducing you to aggregate functions. You learned that there are six different functions and that you can use them in the SELECT and WHERE clauses of a SELECT statement. You also learned that each aggregate function—except COUNT(*)—disregards all Null values as it performs its operation.

Next we showed how to use each aggregate function. You learned how to count rows or values with the COUNT functions, how to find the largest and smallest values with the MAX and MIN functions, how to calculate a mean average with the AVG function, and how to total a set of values with the SUM function. We also showed how to use the DISTINCT option with each function and explained that DISTINCT has no effect on the MAX and MIN functions.

We closed the chapter by showing you how to use aggregate functions in filters. You now know that you can use an aggregate function within a subquery

and then use the subquery as part of the filter. You also learned that you can apply a filter to the subquery as well so that the aggregate function bases its value on a specific set of data.

We've only just begun to show you what you can do with aggregate functions. In the next two chapters, we'll show you how to provide more sophisticated statistical information by using aggregate functions on *grouped* data and how to apply a filter to aggregate calculations.

The following section presents a number of requests that you can work out on your own.

Problems for You to Solve

Below, we show you the request statement and the name of the solution query in the sample databases. If you want some practice, you can work out the SQL you need for each request and then check your answer with the query we saved in the samples. Don't worry if your syntax doesn't exactly match the syntax of the queries we saved—as long as your result set is the same.

Sales Orders Database

1. *"What is the average retail price of a mountain bike?"*
   You can find the solution in CH12_Average_Price_Of_A_Mountain_Bike (1 row).
2. *"What was the date of our most recent order?"*
   You can find the solution in CH12_Most_Recent_Order_Date (1 row).
3. *"What was the total amount for order number 8?"*
   You can find the solution in CH12_Total_Amount_For_Order_Number_8 (1 row).

Entertainment Agency Database

1. *"What is the average salary of a booking agent?"*
   You can find the solution in CH12_Average_Agent_Salary (1 row).
2. *"Show me the engagement numbers for all engagements that have a contract price greater than or equal to the overall average contract price."*
   (Hint: You'll have to use a subquery to answer this request.)
   You can find the solution in CH12_Contract_Price_GE_Average_Contract_Price (45 rows).

3. *"How many of our entertainers are based in Bellevue?"*
   You can find the solution in CH12_Number_Of_Bellevue_Entertainers (1 row).

School Scheduling Database

1. *"What is the current average class duration?"*
   You can find the solution in CH12_Average_Class_Duration (1 row).
2. *"List the last name and first name of each staff member who has been with us since the earliest hire date."*
   (Hint: You'll have to use a subquery containing an aggregate function that evaluates the DateHired column.)
   You can find the solution in CH12_Most_Senior_Staff_Members (3 rows).
3. *"How many classes are held in room 3346?"*
   You can find the solution in CH12_Number_Of_Classes_Held_In_Room_3346 (1 row).

Bowling League Database

1. *"What is the largest handicap held by any bowler at the current time?"*
   You can find the solution in CH12_Current_Highest_Handicap (1 row).
2. *"Which locations hosted tournaments on the earliest tournament date?"*
   You can find the solution in CH12_Tourney_Locations_For_Earliest_Date (2 rows).
3. *"What is the last tournament date we have in our schedule?"*
   You can find the solution in CH12_Last_Tourney_Date (1 row).

Recipes Database

1. *"Which recipe requires the most cloves of garlic?"*
   (Hint: You'll need to use INNER JOINs and a subquery to answer this request.)
   You can find the solution in CH12_Recipe_With_Most_Cloves_of_Garlic (1 row).
2. *"Count the number of main course recipes."*
   (Hint: This requires a JOIN between Recipe_Classes and Recipes.)
   You can find the solution in CH12_Number_Of_Main_Course_Recipes (1 row).
3. *"Calculate the total number of teaspoons of salt in all recipes."*
   You can find the solution in CH12_Total_Salt_Used (1 row).

13

Grouping Data

"Don't drown yourself with details.
Look at the whole."
—Marshal Ferdinand Foch
Commander-in-Chief,
Allied armies in France

Topics Covered in This Chapter

Why Group Data?

The GROUP BY Clause

"Some Restrictions Apply"

Uses for GROUP BY

Sample Statements

Summary

Problems for You to Solve

Chapter 12, Simple Totals, explained how to use the aggregate functions (COUNT, MIN, MAX, AVG, and SUM) to ask SQL to calculate a value across all the rows in the table defined in your FROM and WHERE clauses. We pointed out, however, that after you include any value expression that contains an aggregate function in your SELECT clause, *all* your value expressions must either be a literal constant or contain an aggregate function. This characteristic is useful if you want to see only one row of totals across a result set, but what if you want to see some subtotals? In this chapter, we'll show you how to ask for subtotals by grouping your data.

Why Group Data?

When you're working in the Sales Orders database, finding out the number of orders (COUNT), the total sales (SUM), the average of sales (AVG), the smallest order (MIN), or the largest order (MAX) is useful, indeed. And if you want to calculate any of these values by customer, order date, or product, you can add a filter (WHERE) to fetch the rows for one particular customer or product. But what if you want to see subtotals for *all* customers, displaying the customer name along with the subtotals? To do that, you need to ask your database system to *group* the rows.

Likewise, in the Entertainment Agency database, it's easy to find out the number of contracts, the total contract price, the smallest contract price, or the largest contract price for all contracts. You can even filter the rows so that you see these calculations for one particular entertainer, one particular customer, or across a specific range of dates. Again, if you want to see one total row for each customer or entertainer, you must group the rows.

Are you starting to get the idea? When you ask your database system to group rows on column values or expressions, it forms subsets of rows based on matching values. You can then ask your database to calculate aggregate values *on each group*. Let's look at a simple example from the Entertainment Agency database. First, we need to build a query that fetches the columns of interest—entertainer name and contract price. Here's the SQL.

```
SQL         SELECT Entertainers.EntStageName,
               Engagements.ContractPrice
            FROM Entertainers
            INNER JOIN Engagements
            ON Entertainers.EntertainerID =
               Engagements.EntertainerID
            ORDER BY EntStageName
```

The result looks like the following table. (In the sample database, we saved this request as CH13_Entertainers_And_ContractPrices.)

EntStageName	ContractPrice
Carol Peacock Trio	$140.00
Carol Peacock Trio	$1,670.00
Carol Peacock Trio	$770.00
Carol Peacock Trio	$1,670.00
Carol Peacock Trio	$1,670.00
Carol Peacock Trio	$320.00
Carol Peacock Trio	$1,400.00
Carol Peacock Trio	$680.00
Carol Peacock Trio	$410.00
Carol Peacock Trio	$1,940.00
Carol Peacock Trio	$410.00
Caroline Coie Cuartet	$1,250.00
Caroline Coie Cuartet	$2,450.00
Caroline Coie Cuartet	$1,490.00
Caroline Coie Cuartet	$1,370.00
<< *more rows here* >>	

You already know that you can count all the rows, or find the smallest, largest, sum, or average of the ContractPrice column—as long as you eliminate the EntStageName column. However, you can keep this column if you ask your database to group on it. If you ask to group on entertainer stage name, your database will form one group containing the first eleven rows ("Carol Peacock Trio"), a second group containing the next eleven rows ("Caroline Coie Cuartet"), and so on through the entire table. You can now ask for the COUNT of the rows or the SUM, MIN, MAX, or AVG of the ContractPrice column, and you will get one aggregate row per entertainment group. The result looks like the following table.

EntStageName	NumContracts	TotPrice	MinPrice	MaxPrice	AvgPrice
Carol Peacock Trio	11	$11,080.00	$140.00	$1,940.00	$1,007.27
Caroline Coie Cuartet	11	$15,070.00	$290.00	$2,450.00	$1,370.00
Coldwater Cattle Company	8	$14,875.00	$350.00	$3,800.00	$1,859.38
Country Feeling	15	$34,080.00	$275.00	$14,105.00	$2,272.00
Jazz Persuasion	7	$5,480.00	$500.00	$1,670.00	$782.86
Jim Glynn	9	$3,030.00	$110.00	$770.00	$336.67
Julia Schnebly	8	$4,345.00	$275.00	$875.00	$543.13
JV & the Deep Six	10	$17,150.00	$950.00	$3,650.00	$1,715.00
Modern Dance	10	$14,600.00	$650.00	$2,930.00	$1,460.00
Saturday Revue	9	$11,550.00	$290.00	$2,930.00	$1,283.33
Susan McLain	6	$2,670.00	$230.00	$800.00	$445.00
Topazz	7	$6,620.00	$590.00	$1,550.00	$945.71
<< more rows here >>					

Looks interesting, doesn't it? We bet you'd like to know how we did that! We'll show you all the details in the following sections.

The GROUP BY Clause

As you discovered in Chapter 12, you can find out all sorts of interesting information by using aggregate functions. However, you might have noticed that all the examples we gave you applied the aggregate functions across *all* the rows returned by the FROM and WHERE clauses. You could filter the result set down to one group using the WHERE clause, but there was really no way to look at the results from multiple groups in one request. To accomplish this summarizing by group in a single request, we need to add one more major clause to your SQL vocabulary—GROUP BY.

Syntax

Let's take a close look at the GROUP BY clause. Figure 13–1 shows the basic diagram for a SELECT statement with GROUP BY added.

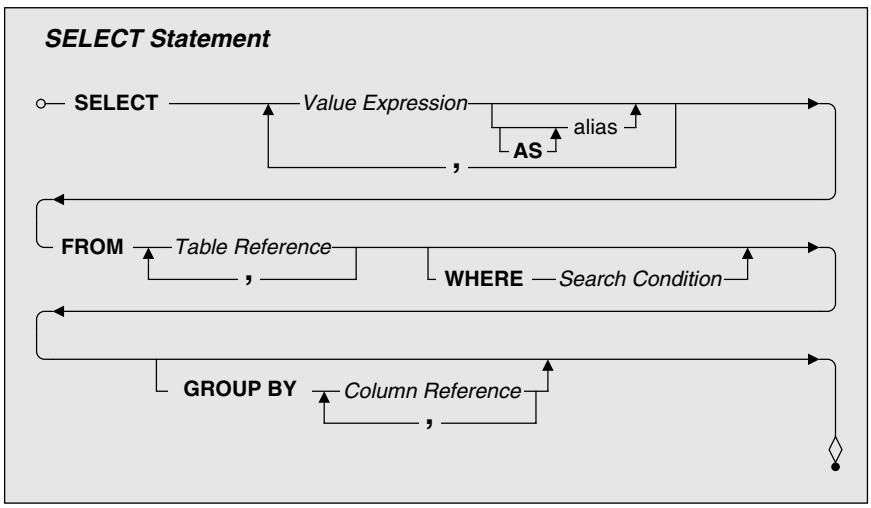


Figure 13–1 *The syntax diagram of a SELECT statement with a GROUP BY clause*

As you recall from earlier chapters, you define the tables that are the source of your data in the FROM clause. Your FROM clause can be as simple as a single table name or as complex as a JOIN of multiple tables. As discussed in Chapter 8, INNER JOINs, you can even embed an entire table subquery (a SELECT statement) as a table reference. Next, you can optionally provide a WHERE clause to include or exclude certain rows supplied by the FROM clause. We covered the WHERE clause in detail in Chapter 6, Filtering Your Data.

When you add a GROUP BY clause, you specify the columns in the logical table formed by the FROM and WHERE clauses that you want your database system to use as the definition for groups of rows. Rows that have the same values in the list of columns you specify will be gathered into a group. You can use the columns that you list in the GROUP BY clause in value expressions in your SELECT clause, and you can use any of the aggregate functions we discussed in the previous chapter to perform calculations across each group.

Let's apply the GROUP BY clause to see how you can calculate information about contract prices by entertainment group—the sample we tantalized you with earlier. Figure 13–2 shows the tables needed to solve this problem.

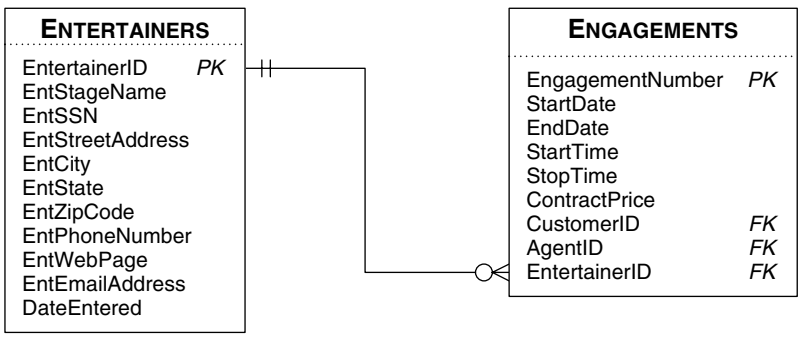


Figure 13–2 *The relationship between the Entertainers and Engagements tables*

❖ **Note** Throughout this chapter, we use the "Request/Translation/Clean Up/SQL" technique introduced in Chapter 4, Creating a Simple Query.

> *"Show me for each entertainment group the group name, the count of contracts for the group, the total price of all the contracts, the lowest contract price, the highest contract price, and the average price of all the contracts."*

(Hint: When you see a request that wants the count, total, smallest, largest, or average of values at a detail level [contracts] *for each* value at a higher level [entertainers], you are going to need to use aggregate functions and grouping in your request. Remember that for each entertainer there are most likely many contracts.)

Translation Select entertainer name, the count of contracts, the sum of the contract price, the lowest contract price, the highest contract price, and the average contract price from the entertainers table joined with the engagements table on entertainer ID, grouped by entertainer name

Clean Up Select entertainer name, ~~the~~ count ~~of~~ (*) ~~contracts~~,
~~the~~ sum ~~of the~~ (contract price),
~~the lowest~~ min(contract price),
~~the highest~~ max(contract price),
~~and the average~~ avg(contract price)
from ~~the~~ entertainers ~~table~~

```
            inner joined with the engagements table
            on entertainers.entertainer ID in the entertainers table
            matches
            = engagements.entertainer ID in the engagements table,
            grouped by entertainer name
SQL         SELECT Entertainers.EntStageName,
               COUNT(*) AS NumContracts,
               SUM(Engagements.ContractPrice) AS TotPrice,
               MIN(Engagements.ContractPrice) AS MinPrice,
               MAX(Engagements.ContractPrice) AS MaxPrice,
               AVG(Engagements.ContractPrice) AS AvgPrice
            FROM Entertainers
            INNER JOIN Engagements
            ON Entertainers.EntertainerID =
               Engagements.EntertainerID
            GROUP BY Entertainers.EntStageName
```

Note that we substituted MIN for "lowest," MAX for "highest," and AVG for "average," as we showed you in the previous chapter. We also asked for COUNT(*) because we want to count all the engagement (contract) rows regardless of any Null values. Adding the GROUP BY clause is what gets us the aggregate calculations *per entertainment group*. It also allows us to include the entertainer name in the SELECT clause. (We saved this request as CH13_Aggregate_Contract_Info_By_Entertainer in the sample database.)

Do you suppose the above query returns a row for each entertainer? What about entertainers who have never been booked? If you remember what you learned in Chapter 9 about OUTER JOIN, you might be tempted to solve the problem like this:

```
SQL         SELECT Entertainers.EntStageName,
               COUNT(*) AS NumContracts,
               SUM(Engagements.ContractPrice) AS TotPrice,
               MIN(Engagements.ContractPrice) AS MinPrice,
               MAX(Engagements.ContractPrice) AS MaxPrice,
               AVG(Engagements.ContractPrice) AS AvgPrice
            FROM Entertainers
            LEFT OUTER JOIN Engagements
            ON Entertainers.EntertainerID =
               Engagements.EntertainerID
            GROUP BY Entertainers.EntStageName
```

One interesting point about all the aggregate functions is that they ignore rows that have a Null value. The above query will return a blank or Null value

for TotPrice, MinPrice, MaxPrice, and AvgPrice for the one entertainer who has no engagements, but you'll find that NumContracts is 1! How can that be? Well, this SQL asks for COUNT(*)—count any row returned. The OUTER JOIN returns exactly one row for the entertainer with no booking, so the count of 1 is correct. But if you remember from the previous chapter, you can also COUNT(*value expression*), and that tells your database system to add to the count only if it finds a non-Null value in the value expression or column name you specify. Let's tweak the query one more time.

```
SQL         SELECT Entertainers.EntStageName,
               COUNT(Engagements.EntertainerID) AS NumContracts,
               SUM(Engagements.ContractPrice) AS TotPrice,
               MIN(Engagements.ContractPrice) AS MinPrice,
               MAX(Engagements.ContractPrice) AS MaxPrice,
               AVG(Engagements.ContractPrice) AS AvgPrice
            FROM Entertainers
            LEFT OUTER JOIN Engagements
            ON Entertainers.EntertainerID =
               Engagements.EntertainerID
            GROUP BY Entertainers.EntStageName
```

Because the EntertainerID column from the Engagements table for the one entertainer who has no bookings is Null, nothing gets counted. If you run this query, you should see the correct value 0 in NumContracts for the one entertainer who has no engagements.

What if you want (or need) to group on more than one value? Let's look at this same problem, but from the perspective of customers rather than entertainers, and let's assume you want to display in your result set both the customer's last name and first name. Figure 13–3 shows the necessary tables.

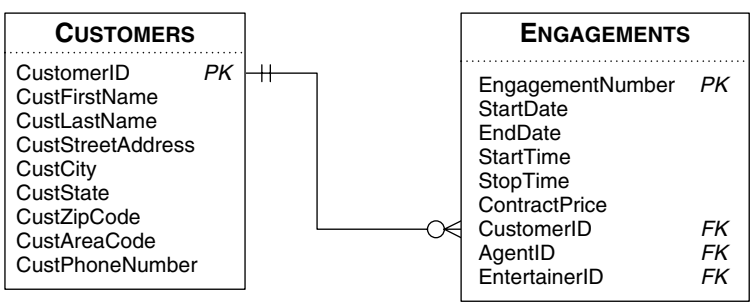


Figure 13–3 *The relationship between the Customers and Engagements tables*

"Show me for each customer the customer first and last names, the count of contracts for the customer, the total price of all the contracts, the lowest contract price, the highest contract price, and the average price of all the contracts."

Translation: Select customer last name, customer first name, the count of contracts, the sum of the contract price, the lowest contract price, the highest contract price, and the average contract price from the customers table joined with the engagements table on customer ID, grouped by customer last name and customer first name

Clean Up:
Select customer last name, customer first name,
~~the~~ count ~~of~~ (*) ~~contracts~~, ~~the~~ sum ~~of the~~ (contract price),
~~the lowest~~ min(contract price), ~~the highest~~ max(contract price),
~~and the average~~ avg(contract price)
from ~~the~~ customers ~~table~~
inner join~~ed with the~~ engagements ~~table~~
on customers.customer ID ~~in the customers table matches~~
= engagements.customer ID ~~in the engagements table,~~
group~~ed~~ by customer last name, ~~and~~ customer first name

SQL:

```
SELECT Customers.CustLastName,
   Customers.CustFirstName,
   COUNT(*) AS NumContracts,
   SUM(Engagements.ContractPrice) AS TotPrice,
   MIN(Engagements.ContractPrice) AS MinPrice,
   MAX(Engagements.ContractPrice) AS MaxPrice,
   AVG(Engagements.ContractPrice) AS AvgPrice
FROM Customers
INNER JOIN Engagements
ON Customers.CustomerID =
   Engagements.CustomerID
GROUP BY Customers.CustLastName,
   Customers.CustFirstName
```

The result looks like the following table. (In the sample database, we saved this request as CH13_Aggregate_Contract_Info_By_Customer.)

CustLastName	CustFirstName	NumContracts	TotPrice	MinPrice	MaxPrice	AvgPrice
Berg	Matt	9	$13,170.00	$200.00	$2,675.00	$1,463.33
Brehm	Peter	7	$7,250.00	$290.00	$3,800.00	$1,035.71
Ehrlich	Zachary	13	$12,455.00	$230.00	$1,550.00	$958.08
Hallmark	Elizabeth	8	$25,585.00	$410.00	$14,105.00	$3,198.13
Hartwig	Doris	8	$10,795.00	$140.00	$2,750.00	$1,349.38
Keyser	Liz	7	$4,685.00	$200.00	$1,490.00	$669.29
McCrae	Dean	11	$11,800.00	$290.00	$2,570.00	$1,072.73
Patterson	Kerry	7	$6,815.00	$110.00	$2,930.00	$973.57
<< more rows here >>						

Because it takes two columns to display the customer name, we had to include them *both* in the GROUP BY clause. Remember that if you want to include a column in the output that is not the result of an aggregate calculation, you must also include it in the GROUP BY clause. We did not include ContractPrice in the GROUP BY clause because that's the column we're using in many of the aggregate function expressions. If we had included ContractPrice, we would have gotten unique groups of customers and prices. MIN, MAX, and AVG will all return that grouped price. COUNT will be greater than one only if more than one contract with the same price exists for a given customer. If you think about it, though, grouping by customer and price and asking for a COUNT would be a good way to find customers who have multiple contracts at the same price.

Do you suppose this query includes customers who have no bookings? If you answered "No," you're correct! To fetch data for all customers regardless of whether they've booked an engagement, you must use an OUTER JOIN and be careful to COUNT one of the columns from the Engagements table. The solution is similar to that discussed earlier for entertainers and engagements.

Mixing Columns and Expressions

Suppose you want to list the customer name as one output column, the full customer address as another output column, the last engagement date, and the sum of engagement contract prices. The customer name is in two columns: CustFirstName and CustLastName. The columns you need for a full

address are CustStreetAddress, CustCity, CustState, and CustZipCode. Let's see how you should construct the SQL for this request. (We saved this request as CH13_Customers_Last_Booking in the sample database.)

"Show me for each customer the customer full name, the customer full address, the latest contract date for the customer, and the total price of all the contracts."

Translation: Select customer last name and customer first name as CustomerFullName; street address, city, state, and ZIP Code as CustomerFullAddress; the latest contract start date; and the sum of the contract price from the customers table joined with the engagements table on customer ID, grouped by customer last name, customer first name, customer street address, customer city, customer state, and customer ZIP Code

Clean Up: Select customer last name ~~and~~ || ',' || customer first name as CustomerFullName, street address~~,~~ || ',' || city~~,~~ || ',' || state~~,~~ ~~and~~ || ' ' || ZIP Code as CustomerFullAddress,
~~the latest~~ max(contract start date) as latest date,
~~and the~~ sum ~~of the~~ (contract price) as total contract price
from ~~the~~ customers ~~table~~
inner join~~ed with the~~ engagements ~~table~~
on customers.customer ID ~~in the customers table matches~~ = engagements.customer ID ~~in the engagements table~~
group~~ed~~ by customer last name, customer first name, customer street address, customer city, customer state, ~~and~~ customer ZIP Code

SQL:

```
SELECT Customers.CustLastName || ', ' ||
   Customers.CustFirstName AS CustomerFullName,
   Customers.CustStreetAddress || ', ' ||
   Customers.CustCity || ', ' ||
   Customers.CustState || ' ' ||
   Customers.CustZipCode AS CustomerFullAddress
   MAX(Engagements.StartDate) AS LatestDate,
   SUM(Engagements.ContractPrice),
   AS TotalContractPrice
FROM Customers
INNER JOIN Engagements
ON Customers.CustomerID =
   Engagements.CustomerID
GROUP BY Customers.CustLastName,
   Customers.CustFirstName,
```

```
Customers.CustStreetAddress,
Customers.CustCity, Customers.CustState,
Customers.CustZipCode
```

Notice that we had to list each and every one of the columns we used in an output expression that did not include an aggregate function. We used StartDate and ContractPrice in aggregate expressions, so we don't need to list them in the GROUP BY clause. In fact, it doesn't make sense to group on either StartDate or ContractPrice because we want to use these in an aggregate calculation across multiple customers. If, for example, we grouped on StartDate, MAX(StartDate) would return the grouping value, and the expression SUM(ContractPrice) would return only the sum of contract prices for a customer on any given date. You wouldn't get the sum of more than one contract unless a customer had more than one contract for a given date—not likely.

Using GROUP BY in a Subquery in a WHERE Clause

In Chapter 11, Subqueries, we introduced the COUNT and MAX aggregate functions to show how to filter rows using an aggregate value fetched with a subquery. In Chapter 12, Simple Totals, we showed how to use MIN, AVG, and SUM in a subquery filter as well. Let's look at a request that requires both a subquery with an aggregate function and a GROUP BY clause in the subquery.

"Display the engagement contract whose price is greater than the sum of all contracts for any other customer."

Translation

Select customer first name, customer last name, engagement start date, and engagement contract price from the customers table joined with the engagements table on customer ID where the contract price is greater than the sum of all contract prices from the engagements table for customers other than the current customer, grouped by customer ID

Clean Up

Select customer first name, customer last name,
engagement start date, ~~and~~ engagement contract price
from ~~the~~ customers ~~table~~
inner join~~ed with the~~ engagements ~~table~~
on customers.customer ID ~~in the customers table matches~~
= engagements.customer ID ~~in the engagements table~~
where ~~the~~ contract price ~~is greater than~~ > ALL
(select ~~the~~ sum ~~of all~~ (contract price~~s~~)
from ~~the~~ engagements ~~table~~ as E2

~~for~~ where E2.customer~~s~~ ID <>
~~other than the current~~ customers.customer ID~~;~~
group~~ed~~ by E2.customer ID)

SQL

```
SELECT Customers.CustFirstName,
   Customers.CustLastName,
   Engagements.StartDate,
   Engagements.ContractPrice
FROM Customers
INNER JOIN Engagements
ON Customers.CustomerID =
   Engagements.CustomerID
WHERE Engagements.ContractPrice > ALL
   (Select SUM(ContractPrice)
   FROM Engagements AS E2
   WHERE E2.CustomerID <> Customers.CustomerID
   GROUP BY E2.CustomerID)
```

Let's analyze what the subquery is doing. For each engagement that the query looks at in the JOIN of the Customers and Engagements tables, the subquery calculates the SUM of all contract prices for all *other* customers and groups them by customer ID. Because there are multiple customers in the database, the subquery will return multiple SUM values—one for each of the other customers. For this reason, we cannot ask for a simple greater than (>) comparison. We can, however, use the quantified greater than all (> ALL) comparison to check a set of values as you learned in Chapter 11. If you run this query in the sample Entertainment Agency database for this chapter (we saved it as CH13_Biggest_Big_Contract), you'll find that one contract fits the bill, as shown here.

CustFirstName	CustLastName	StartDate	ContractPrice
Elizabeth	Hallmark	2008-01-22	$14,105.00

Simulating a SELECT DISTINCT Statement

Did it occur to you that you can use a GROUP BY clause and not include any aggregate functions in your SELECT clause? Sure you can! When you do this, you get the same effect as using the DISTINCT keyword covered in Chapter 4. (See the Eliminating Duplicate Rows section in that chapter.)

Let's look at a simple request that requires unique values and solve it using both techniques.

"Show me the unique city names from the customers table."

Translation 1	Select the unique city names from the customers table
Clean Up	Select ~~the unique~~ distinct city names from ~~the~~ customers ~~table~~
SQL	`SELECT DISTINCT Customers.CustCityName` `FROM Customers`
Translation 2	Select city name from the customers table, grouped by city name
Clean Up	Select city name from ~~the~~ customers ~~table,~~ group~~ed~~ by city name

SQL	`SELECT Customers.CustCityName`
	`FROM Customers`
	`GROUP BY Customers.CustCityName`

Remember that GROUP BY groups all the rows on the grouping column(s) you specify and returns one row per group. This is a slightly different way to get to the same result that you obtain with the DISTINCT keyword. Which one is better? We think that DISTINCT might be a clearer statement of what you want if all you want is unique rows, but you might find that your database system solves the problem faster when you use GROUP BY. In addition, GROUP BY lets you obtain more information about your data. Consider the following query:

```
SQL         SELECT Customers.CustCityName, Count(*) as
            CustPerCity
            FROM Customers
            GROUP BY Customers.CustCityName
```

With this query, you not only fetch the unique city names but also find out how many customers are in each city. Is that cool or what?

"Some Restrictions Apply"

We already mentioned that adding a GROUP BY clause places certain restrictions on constructing your request. Let's review those restrictions to make sure you don't fall into common traps.

Column Restrictions

When you add a GROUP BY clause, you're asking your database system to form unique groups of rows from those returned by the tables defined in your FROM clause and filtered by your WHERE clause. You can use as many aggregate expressions as you like in your SELECT clause, and these expressions can use any of the columns in the table defined by the FROM and WHERE clauses. As we pointed out in an earlier example, it probably does not make sense to reference a column in an aggregate expression and also include that column in your grouping specification.

If you choose to also include expressions that reference columns but do not include an aggregate function, you must list *all* columns you use this way in the GROUP BY clause. One of the most common mistakes is to assume that you can reference columns in nonaggregate expressions as long as the columns come from unique rows. For example, let's look at an incorrect request that includes a primary key value—something that we know by definition is unique.

"Display the customer ID, customer full name, and the total of all engagement contract prices."

Translation: Select customer ID, customer first name, and customer last name as CustFullName, and the sum of contract prices as TotalPrice from the customers table joined with the engagements table on customer ID, grouped by customer ID

Clean Up:
Select customer ID, customer first name
~~and~~ || ' ' || customer last name as CustFullName,
~~and the~~ sum ~~of~~ (contract price)~~s~~ as TotalPrice
from ~~the~~ customers ~~table~~
inner join~~ed with the~~ engagements ~~table~~
on customers.customer ID ~~in the customers table matches~~
= engagements.customer ID ~~in the engagements table,~~
group~~ed~~ by customer ID

SQL:

```
SELECT Customers.CustomerID,
   Customers.CustFirstName || ' ' ||
   Customers.CustLastName AS CustFullName,
   SUM(Engagements.ContractPrice) AS TotalPrice
FROM Customers
INNER JOIN Engagements
ON Customers.CustomerID =
   Engagements.CustomerID
GROUP BY Customers.CustomerID
```

We *know* that CustomerID is unique per customer. Grouping on CustomerID alone should be sufficient to fetch unique customer first and last name information within the groups formed by CustomerID. However, SQL is a language based on syntax, not semantics. In other words, SQL does not take into account any knowledge that could be implied by the design of your database tables—including whether columns are primary keys. SQL demands that your request be syntactically "pure" and translatable without any knowledge of the underlying table design. So, the above SQL statement will fail on a database system that is fully compliant with the SQL Standard because we've included columns in the SELECT clause that are not in an aggregate function and are also not in the GROUP BY clause (CustFirstName and CustLastName). The correct SQL request is as follows.

```
SQL         SELECT Customers.CustomerID,
               Customers.CustFirstName || ' ' ||
               Customers.CustLastName AS CustFullName,
               SUM(Engagements.ContractPrice) AS TotalPrice
            FROM Customers
            INNER JOIN Engagements
            ON Customers.CustomerID =
               Engagements.CustomerID
            GROUP BY Customers.CustomerID,
               Customers.CustFirstName,
               Customers.CustLastName
```

This might seem like overkill, but it's the correct way to do it!

❖ **Note** In some database systems, you must exactly duplicate the *expressions* you use in the SELECT clause in the GROUP BY clause. Oracle and Microsoft Office Access are examples of systems that require this. In our example, you would have to end the SQL with this:

```
            GROUP BY Customers.CustomerID,
               Customers.CustFirstName || ' ' ||
               Customers.CustLastName
```

instead of listing the separate columns. This isn't compliant with the SQL Standard, but you might find that this is the only way you can get your request to work on your system.

Grouping on Expressions

We showed you earlier some correct examples of creating expressions that do not include aggregate functions. One of the most common mistakes is to attempt to group on the expression you create in the SELECT clause rather than on the individual columns. Remember that the GROUP BY clause must refer to columns created by the FROM and WHERE clauses. It cannot use an expression you create in your SELECT clause.

Let's take another look at an example we solved earlier to show you what we mean, but this time, let's make the mistake. (We're skipping the Translation and Clean Up steps here because we covered them earlier.)

> *"Show me for each customer in the state of Washington the customer full name, the customer full address, the latest contract date for the customer, and the total price of all the contracts."*

```
SQL         SELECT Customers.CustLastName || ', ' ||
               Customers.CustFirstName AS CustomerFullName,
               Customers.CustStreetAddress || ', ' ||
               Customers.CustCity || ', ' ||
               Customers.CustState || ' ' ||
               Customers.CustZip AS CustomerFullAddress
               MAX(Engagements.StartDate) AS LatestDate,
               SUM(Engagements.ContractPrice)
               AS TotalContractPrice
            FROM Customers
            INNER JOIN Engagements
            ON Customers.CustomerID =
               Engagements.CustomerID
            WHERE Customers.CustState ='WA'
            GROUP BY CustomerFullName,
               CustomerFullAddress
```

Some database systems will let you get away with this, but it's not correct. The CustomerFullName and CustomerFullAddress columns don't exist until *after* your database system has evaluated the FROM, WHERE, and GROUP BY clauses. The GROUP BY clause won't find these columns in the result created in the FROM and WHERE clauses, so on a database system that strictly adheres to the SQL Standard you'll get a syntax error.

We showed you earlier one correct way to solve this: You must list all the columns you use in both the CustomerFullName and CustomerFullAddress expressions. Another way is to make the FROM clause generate the calculated columns by embedding a table subquery. Here's what it looks like.

```
SQL         SELECT CE.CustomerFullName,
               CE.CustomerFullAddress,
               MAX(CE.StartDate) AS LatestDate,
               SUM(CE.ContractPrice)
               AS TotalContractPrice
            FROM
              (SELECT Customers.CustLastName || ', ' ||
               Customers.CustFirstName AS CustomerFullName,
                  Customers.CustStreetAddress || ', ' ||
                  Customers.CustCity || ', ' ||
                  Customers.CustState || ' ' ||
                  Customers.CustZip AS CustomerFullAddress,
                  Engagements.StartDate,
                  Engagements.ContractPrice
               FROM Customers
               INNER JOIN Engagements
               ON Customers.CustomerID =
                 Engagements.CustomerID
               WHERE Customers.CustState ='WA')
            AS CE
            GROUP BY CE.CustomerFullName,
               CE.CustomerFullAddress
```

This works now because we've generated the CustomerFullName and CustomerFullAddress columns as output in the FROM clause. You have to admit, though, that this makes the query very complex. In truth, it's better to just list all the individual columns you plan to use in nonaggregate expressions rather than try to generate the expressions as columns inside the FROM clause.

Uses for GROUP BY

At this point, you should have a fairly good understanding of how to ask for subtotals across groups using aggregate functions and the GROUP BY clause. The best way to give you an idea of the wide range of uses for GROUP BY is to list some problems you can solve with this new clause and then present a robust set of examples in the Sample Statements section.

"Show me each vendor and the average by vendor of the number of days to deliver products."

"Display for each product the product name and the total sales."

"List for each customer and order date the customer full name and the total cost of items ordered on each date."

"Display each entertainment group ID, entertainment group member, and the amount of pay for each member based on the total contract price divided by the number of members in the group."

"Show each agent name, the sum of the contract price for the engagements booked, and the agent's total commission."

"For completed classes, list by category and student the category name, the student name, and the student's average grade of all classes taken in that category."

"Display by category the category name and the count of classes offered."

"List each staff member and the count of classes each is scheduled to teach."

"Show me for each tournament and match the tournament ID, the tournament location, the match number, the name of each team, and the total of the handicap score for each team."

"Display for each bowler the bowler name and the average of the bowler's raw game scores."

"Show me how many recipes exist for each class of ingredient."

"If I want to cook all the recipes in my cookbook, how much of each ingredient must I have on hand?"

Sample Statements

You now know the mechanics of constructing queries using a GROUP BY clause and have seen some of the types of requests you can answer. Let's take a look at a set of samples, all of which request that the information be grouped. These examples come from each of the sample databases.

We've also included sample result sets that would be returned by these operations and placed them immediately after the SQL syntax line. The name that appears immediately above a result set is the name we gave each query in the sample data on the companion CD you'll find bound into the back of the book. We stored each query in the appropriate sample database (as indicated within the example), and we prefixed the names of the queries relevant to this chapter with "CH13." You can follow the instructions in the Introduction of this book to load the samples onto your computer and try them.

❖ **Note** Remember that all the column names and table names used in these examples are drawn from the sample database structures shown in Appendix B, Schema for the Sample Databases. To simplify the process, we have combined the Translation and Clean Up steps for all the examples.

These samples assume you have thoroughly studied and understood the concepts covered in previous chapters, especially the chapters on JOINs and subqueries.

Sales Orders Database

"List for each customer and order date the customer full name and the total cost of items ordered on each date."

Translation/ Clean Up

Select customer first name ~~and~~ || ' ' || customer last name as CustFullName, order date, ~~and the~~
sum ~~of~~ (quoted price ~~times~~ * quantity ordered) as TotalCost
from ~~the~~ customers ~~table~~
inner join~~ed with the~~ orders ~~table~~
on customers.customer ID ~~in the customers table matches~~
= orders.customer ID ~~in the orders table,~~
~~and then inner~~ join~~ed with the~~ order details ~~table~~
on orders.order number ~~in the orders table matches~~
= order_details.order number ~~in the order details table,~~
group~~ed~~ by customer first name,
customer last name, ~~and~~ order date

SQL

```
SELECT Customers.CustFirstName || ' ' ||
   Customers.CustLastName AS CustFullName,
   Orders.OrderDate,
   SUM(Order_Details.QuotedPrice *
   Order_Details.QuantityOrdered) AS TotalCost
FROM (Customers
INNER JOIN Orders
ON Customers.CustomerID = Orders.CustomerID)
INNER JOIN Order_Details
ON Orders.OrderNumber =
   Order_Details.OrderNumber
GROUP BY Customers.CustFirstName,
   Customers.CustLastName, Orders.OrderDate
```

CH13_Order_Totals_By_Customer_And_Date (847 rows)

CustFullName	OrderDate	TotalCost
Alaina Hallmark	2007-09-02	$4,699.98
Alaina Hallmark	2007-09-14	$4,433.95
Alaina Hallmark	2007-09-21	$353.25
Alaina Hallmark	2007-09-22	$3,951.90
Alaina Hallmark	2007-09-30	$10,388.68
Alaina Hallmark	2007-10-12	$3,088.00
Alaina Hallmark	2007-10-22	$6,775.06
Alaina Hallmark	2007-10-30	$15,781.10
<< more rows here >>		

Entertainment Agency Database

"Display each entertainment group ID, entertainment group member, and the amount of pay for each member based on the total contract price divided by the number of members in the group."

❖ **Note** This one is really tricky because each member might belong to more than one entertainer group. You must sum the contract prices for each entertainer and then divide by the count of members in that group (assuming each member gets equal pay). Fetching the count requires a subquery filtered on the current entertainer ID (the ID of the group, not the ID of the member), which means you also must group on entertainer ID. Oh yes, and don't forget to exclude members who are not active (Status = 3).

Translation/ Clean Up

Select entertainer ID, member first name, member last name,
~~and the~~ sum ~~of~~ (contract price)~~s~~ ~~divided by~~ / ~~the~~
(select count(*) ~~of active members~~
from entertainer members as EM2 ~~in the current entertainer group~~
where status ~~is not equal to~~ <> ~~not active~~ 3
and ~~the~~ EM2 ~~table~~ entertainer ID ~~equals~~
= ~~the~~ entertainer members ~~table~~ entertainer ID)
from ~~the~~ members ~~table~~
inner join~~ed with the~~ entertainer members ~~table~~
on members.member ID ~~in the members table matches~~
= entertainer_members.member ID
~~in the entertainer members table,~~
~~then~~ inner join~~ed with the~~ entertainers ~~table~~
on entertainers.entertainer ID ~~in the entertainers table matches~~
= entertainer_members.entertainer ID
~~in the entertainer members table,~~
~~and finally~~ inner join~~ed with the~~ engagements ~~table~~
on entertainers.entertainer ID ~~in the entertainers table matches~~
= engagements.entertainer ID ~~in the engagements table,~~
where member status ~~is not equal to~~ <> ~~not active~~ 3~~,~~
group~~ed~~ by entertainer ID,
member first name, ~~and~~ member last name,
~~sorted~~ order by member last name

```
SQL         SELECT Entertainers.EntertainerID,
               Members.MbrFirstName, Members.MbrLastName,
               SUM(Engagements.ContractPrice)/
                  (SELECT COUNT(*)
                   FROM Entertainer_Members AS EM2
                   WHERE EM2.Status <> 3
                   AND EM2.EntertainerID =
                   Entertainers.EntertainerID)
                AS MemberPay
            FROM ((Members
            INNER JOIN Entertainer_Members
            ON Members.MemberID =
               Entertainer_Members.MemberID)
            INNER JOIN Entertainers
            ON Entertainers.EntertainerID =
               Entertainer_Members.EntertainerID)
            INNER JOIN Engagements
            ON Entertainers.EntertainerID =
               Engagements.EntertainerID
            WHERE Entertainer_Members.Status<>3
            GROUP BY Entertainers.EntertainerID,
               Members.MbrFirstName, Members.MbrLastName
            ORDER BY Members.MbrLastName
```

CH13_Member_Pay (39 rows)

EntertainerID	MbrFirstName	MbrLastName	MemberPay
1010	Kendra	Bonnicksen	$2,887.50
1013	Kendra	Bonnicksen	$3,767.50
1007	Robert	Brown	$2,975.00
1008	Robert	Brown	$6,816.00
1008	George	Chavez	$6,816.00
1013	George	Chavez	$3,767.50
1010	Caroline	Coie	$2,887.50
1013	Caroline	Coie	$3,767.50
<< more rows here >>			

School Scheduling Database

"For completed classes, list by category and student the category name, the student name, and the student's average grade of all classes taken in that category."

Translation/ Clean Up

Select category description, student first name, student last name,
~~and the average~~ AVG(~~of~~ grade) as AvgOfGrade
from ~~the~~ categories ~~table~~
inner join~~ed with the~~ subjects ~~table~~
on categories.category ID ~~in the categories table matches~~
= subjects.category ID ~~in the subjects table,~~
~~then~~ inner join~~ed with the~~ classes ~~table~~
on subjects.subject ID ~~in the subjects table matches~~
= classes.subject ID ~~in the classes table,~~
~~then~~ inner join~~ed with the~~ student schedules ~~table~~
on classes.class ID ~~in the classes table matches~~
= student_schedules.class ID ~~in the student schedules table,~~
~~then~~ inner join~~ed with the~~ student class status ~~table~~
on student_class_status.class status
~~in the student class status table matches~~
= student_schedules.class status ~~in the student schedules table,~~
~~and finally~~ inner join~~ed with the~~ students ~~table~~
on students.student ID ~~in the students table matches~~
= student_schedules.student ID ~~in the student schedules table~~
where class status description ~~is~~ = 'Completed~~,~~'
group~~ed~~ by category description, student first name,
~~and~~ student last name

SQL

```
SELECT Categories.CategoryDescription,
   Students.StudFirstName,
   Students.StudLastName,
   AVG(Student_Schedules.Grade) AS AvgOfGrade
FROM ((((Categories
INNER JOIN Subjects
ON Categories.CategoryID = Subjects.CategoryID)
INNER JOIN Classes
ON Subjects.SubjectID = Classes.SubjectID)
INNER JOIN Student_Schedules
ON Classes.ClassID = Student_Schedules.ClassID)
INNER JOIN Student_Class_Status
ON Student_Class_Status.ClassStatus =
   Student_Schedules.ClassStatus)
INNER JOIN Students
ON Students.StudentID =
   Student_Schedules.StudentID
```

```
WHERE Student_Class_Status.ClassStatusDescription =
   'Completed'
GROUP BY Categories.CategoryDescription,
   Students.StudFirstName,
   Students.StudLastName
```

CH13_Student_GradeAverage_By_Category (47 rows)

CategoryDescription	StudFirstName	StudLastName	AvgOfGrade
Accounting	Betsy	Stadick	90.67
Accounting	David	Hamilton	79.43
Accounting	Elizabeth	Hallmark	90.24
Accounting	John	Kennedy	71.45
Accounting	Michael	Viescas	90.01
Accounting	Sara	Sheskey	89.92
Accounting	Scott	Bishop	87.82
Accounting	Steve	Pundt	84.37
Art	Elizabeth	Hallmark	86.43
Art	George	Chavez	77.45
<< more rows here >>			

Bowling League Database

"Show me for each tournament and match the tournament ID, the tournament location, the match number, the name of each team, and the total of the handicap score for each team."

Translation/ Clean Up

Select tourney ID, tourney location, match ID, team name, ~~and~~ ~~the~~ sum ~~of~~ (handicap score) as TotHandiCapScore
from ~~the~~ tournaments ~~table~~
inner join~~ed with the~~ tourney matches ~~table~~
on tournaments.tourney ID ~~in the tournaments table matches~~
= tourney_matches.tourney ID ~~in the tourney matches table, then~~ inner join~~ed with the~~ match games ~~table~~
on tourney_matches.match ID
~~in the tourney matches table matches~~
= match_games.match ID ~~in the match games table, then~~ inner join~~ed with the~~ bowler scores ~~table~~
on match_games.match ID ~~in the match games table matches~~
= bowler_scores.match ID ~~in the bowler scores table~~
and match_games.game number
~~in the match games table matches~~
= bowler_scores.game number ~~in the bowler scores table, then~~ inner join~~ed with the~~ bowlers ~~table~~
on bowlers.bowler ID ~~in the bowlers table matches~~
= bowler_scores.bowler ID ~~in the bowler scores table, and finally~~ inner join~~ed with the~~ teams ~~table~~
on teams.team ID ~~in the teams table matches~~
= bowlers.team ID ~~in the bowlers table,~~
group~~ed~~ by tourney ID, tourney location, match ID,
~~and~~ team name

SQL

```
SELECT Tournaments.TourneyID,
   Tournaments.TourneyLocation,
   Tourney_Matches.MatchID, Teams.TeamName,
   SUM(Bowler_Scores.HandiCapScore)
   AS TotHandiCapScore
FROM ((((Tournaments
INNER JOIN Tourney_Matches
ON Tournaments.TourneyID =
   Tourney_Matches.TourneyID)
INNER JOIN Match_Games
ON Tourney_Matches.MatchID =
   Match_Games.MatchID)
INNER JOIN Bowler_Scores
ON (Match_Games.MatchID =
   Bowler_Scores.MatchID) AND
```

```
   (Match_Games.GameNumber =
   Bowler_Scores.GameNumber))
INNER JOIN Bowlers
ON Bowlers.BowlerID = Bowler_Scores.BowlerID)
INNER JOIN Teams
ON Teams.TeamID = Bowlers.TeamID
GROUP BY Tournaments.TourneyID,
   Tournaments.TourneyLocation,
   Tourney_Matches.MatchID, Teams.TeamName
```

As you can see, the difficult part of this request is assembling the complex JOIN clauses to link all the tables in the correct manner.

CH13_Tournament_Match_Team_Results (112 rows)

TourneyID	TourneyLocation	MatchID	TeamName	TotHandiCapScore
1	Red Rooster Lanes	1	Marlins	2351
1	Red Rooster Lanes	1	Sharks	2348
1	Red Rooster Lanes	2	Barracudas	2289
1	Red Rooster Lanes	2	Terrapins	2391
1	Red Rooster Lanes	3	Dolphins	2389
1	Red Rooster Lanes	3	Orcas	2395
1	Red Rooster Lanes	4	Manatees	2292
1	Red Rooster Lanes	4	Swordfish	2353
2	Thunderbird Lanes	5	Marlins	2297
2	Thunderbird Lanes	5	Terrapins	2279
<< more rows here >>				

"Display the highest raw score for each bowler."

Translation/ Clean Up

Select bowler first name, bowler last name, ~~and~~
~~the~~ max~~imum~~ (raw score) as HighScore
from ~~the~~ bowlers ~~table~~
inner join~~ed with the~~ bowler scores ~~table~~
on bowlers.bowler ID ~~in the bowlers table matches~~
= bowler_scores.bowler ID ~~in the bowler scores table,~~
group~~ed~~ by bowler first name, ~~and~~ bowler last name

SQL

```
SELECT Bowlers.BowlerFirstName,
   Bowlers.BowlerLastName,
   MAX(Bowler_Scores.RawScore) AS HighScore
FROM Bowlers
INNER JOIN Bowler_Scores
ON Bowlers.BowlerID = Bowler_Scores.BowlerID
GROUP BY Bowlers.BowlerFirstName,
   Bowlers.BowlerLastName
```

CH13_Bowler_High_Score_Group (32 rows)

BowlerFirstName	BowlerLastName	HighScore
Alaina	Hallmark	180
Aslatair	Black	164
Angel	Kennedy	194
Ann	Patterson	165
Bailey	Hallmark	164
Barbara	Fournier	164
Caleb	Viescas	193
Carol	Viescas	150
David	Cunningham	180
David	Fournier	178
<< more rows here >>		

Recipes Database

"Show me how many recipes exist for each class of ingredient."

> ❖ **Note** The challenge here is that you don't want to count a particular recipe class more than once per recipe. For example, if a recipe contains multiple herbs or dairy ingredients, that recipe should be counted only once per class. Sounds like it's time to use COUNT(DISTINCT *value expression*), doesn't it?

Translation/ Clean Up

Select ingredient class description,
~~and the unique~~ count ~~of~~ (distinct recipe ID) as CountOfRecipeID
from ~~the~~ ingredient classes ~~table~~
inner join~~ed with the~~ ingredients ~~table~~
on ingredient_classes.ingredient class ID
~~in the ingredient classes table matches~~
= ingredients.ingredient class ID ~~in the ingredients table, and then~~ inner join~~ed with the~~ recipe ingredients ~~table~~
on ingredients.ingredient ID ~~in the ingredients table matches~~
= recipe_ingredients.ingredient ID
~~in the recipe ingredients table,~~
group~~ed~~ by ingredient class description

SQL

```
SELECT
   Ingredient_Classes.IngredientClassDescription,
   Count(DISTINCT RecipeID) AS CountOfRecipeID
FROM (Ingredient_Classes
INNER JOIN Ingredients
ON Ingredient_Classes.IngredientClassID =
   Ingredients.IngredientClassID)
INNER JOIN Recipe_Ingredients
ON Ingredients.IngredientID =
   Recipe_Ingredients.IngredientID
GROUP BY
   Ingredient_Classes.IngredientClassDescription
```

CH13_IngredientClass_Distinct_Recipe_Count (19 rows)

IngredientClassDescription	CountOfRecipeID
Bottle	1
Butter	3
Cheese	2
Chips	1
Condiment	2
Dairy	2
Fruit	1
Grain	2
Herb	1
<< more rows here >>	

❖ **Note** Because Microsoft Access does not support COUNT DISTINCT, you'll find that the query in the Access sample database first selects the DISTINCT values of RecipeID using a table subquery in the FROM clause and then counts the resulting rows.

SUMMARY

We began the chapter by explaining to you why you might want to group data to get multiple subtotals from a result set. After tantalizing you with an example, we proceeded to show how to use the GROUP BY clause to solve the example problem and several others. We also showed how to mix column expressions with aggregate functions.

We next explored an interesting example of using GROUP BY in a subquery that acts as a filter in a WHERE clause. We subsequently pointed out that constructing a query using GROUP BY and no aggregate functions is the same as using DISTINCT in your SELECT clause. Then we warned you to carefully construct your GROUP BY clause to include the columns and not the expressions.

We wrapped up our discussion of the GROUP BY clause by explaining some common pitfalls. We showed that SQL does not consider any knowledge of

primary keys. We also explained common mistakes you might make when using column expressions in your SELECT clause.

We summarized why the GROUP BY clause is useful and gave you a sample list of problems you can solve using GROUP BY. The rest of the chapter provided examples of how to build requests that require the GROUP BY clause.

The following section presents a number of requests that you can work out on your own.

Problems for You to Solve

Below, we show you the request statement and the name of the solution query in the sample databases. If you want some practice, you can work out the SQL you need for each request and then check your answer with the query we saved in the samples. Don't worry if your syntax doesn't exactly match the syntax of the queries we saved—as long as your result set is the same.

Sales Orders Database

1. *"Show me each vendor and the average by vendor of the number of days to deliver products."*
   (Hint: Use the AVG aggregate function and group on vendor.)
   You can find the solution in CH13_Vendor_Avg_Delivery (10 rows).
2. *"Display for each product the product name and the total sales."*
   (Hint: Use SUM with a calculation of quantity times price and group on product name.)
   You can find the solution in CH13_Sales_By_Product (38 rows).
3. *"List all vendors and the count of products sold by each."*
   You can find the solution in CH13_Vendor_Product_Count_Group (10 rows).
4. Challenge: Now solve problem 3 by using a subquery.
   You can find the solution in CH13_Vendor_Product_Count_Subquery (10 rows).

Entertainment Agency Database

1. *"Show each agent's name, the sum of the contract price for the engagements booked, and the agent's total commission."*
   (Hint: You must multiply the sum of the contract prices by the agent's commission. Be sure to group on the commission rate as well!)
   You can find the solution in CH13_Agent_Sales_And_Commissions (8 rows).

School Scheduling Database

1. *"Display by category the category name and the count of classes offered."*
   (Hint: Use COUNT and group on category name.)
   You can find the solution in CH13_Category_Class_Count (16 rows).
2. *"List each staff member and the count of classes each is scheduled to teach."*
   (Hint: Use COUNT and group on staff name.)
   You can find the solution in CH13_Staff_Class_Count (23 rows).
3. Challenge: Now solve problem 2 by using a subquery.
   You can find the solution in CH13_Staff_Class_Count (27 rows).
4. Can you explain why the subquery solution returns 4 more rows?

Bowling League Database

1. *"Display for each bowler the bowler name and the average of the bowler's raw game scores."*
   (Hint: Use the AVG aggregate function and group on bowler name.)
   You can find the solution in CH13_Bowler_Averages (32 rows).
2. *"Calculate the current average and handicap for each bowler."*
   (Hint: This is a "friendly" league, so the handicap is 90 percent of 200 minus the raw average. Be sure to round the raw average and convert it to an integer before subtracting it from 200, and then round and truncate the final result. Although the SQL Standard doesn't define a ROUND function, most commercial database systems provide one. Check your product documentation for details.)
   You can find the solution in CH13_Bowler_Average_Handicap (32 rows).
3. Challenge: *"Display the highest raw score for each bowler,"* but solve it by using a subquery.
   You can find the solution in CH13_Bowler_High_Score_Subquery (32 rows).

Recipes Database

1. *"If I want to cook all the recipes in my cookbook, how much of each ingredient must I have on hand?"*
   (Hint: Use SUM and group on ingredient name and measurement description.)
   You can find the solution in CH13_Total_Ingredients_Needed (65 rows).
2. *"List all meat ingredients and the count of recipes that include each one."*
   You can find the solution in CH13_Meat_Ingredient_Recipe_Count_Group (4 rows).
3. Challenge: Now solve problem 2 by using a subquery.
   You can find the solution in CH13_Meat_Ingredient_Recipe_Count_Subquery (11 rows).
4. Can you explain why the subquery solution returns 7 more rows?

14

Filtering Grouped Data

"Let schoolmasters puzzle their brain,
With grammar, and nonsense, and learning;
Good liquor, I stoutly maintain,
Gives genius a better discerning."
—Oliver Goldsmith

Topics Covered in This Chapter

A New Meaning of "Focus Groups"

When You Filter Makes a Difference

Uses for HAVING

Sample Statements

Summary

Problems for You to Solve

In Chapter 12, Simple Totals, we gave you the details about all the aggregate functions defined in the SQL Standard. We followed that up in Chapter 13, Grouping Data, with a discussion of how to ask your database system to group sets of rows and then calculate aggregate values on each group. One of the advantages to grouping is that you can also display value expressions based on the grouping columns to identify each group.

In this chapter, we'll put the final piece of the summarizing and grouping puzzle into place. After you group rows and calculate aggregate values, it's often useful to filter further the final result using a predicate on an aggregate calculation. As you'll soon see, you need the last piece of this puzzle—the HAVING clause—to do that.

A New Meaning of "Focus Groups"

You now know that once you've gathered your information into groups of rows, you can request the MIN, MAX, AVG, SUM, or COUNT of all the values in each group. Suppose you want to refine further the final result set—to focus the groups—by testing one of the aggregate values. Let's take a look at a simple request.

> *"Show me the entertainer groups that play in a jazz style and have more than three members."*

Doesn't sound too difficult, does it? Figure 14–1 shows the tables needed to solve this request.

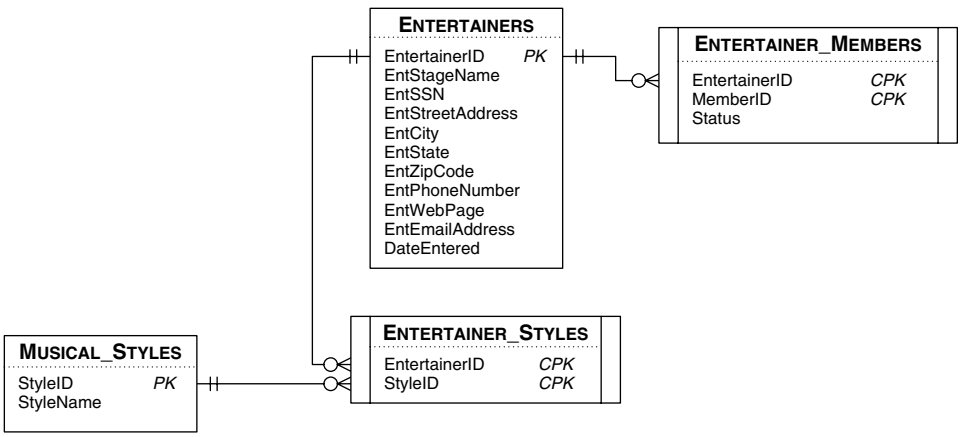


Figure 14–1 *The tables needed to figure out which entertainers play jazz and also have more than three members*

❖ **Note** We again use the "Request/Translation/Clean Up/SQL" technique introduced in Chapter 4, Creating a Simple Query. We also use some JOIN and subquery techniques you learned in Chapter 8, INNER JOINs; Chapter 9, OUTER JOINs; and Chapter 11, Subqueries.

Without knowing about the HAVING clause, you'd probably be tempted to solve it in the following incorrect manner.

Translation Select the entertainer stage name and the count of members from the entertainers table joined with the entertainer members

table on entertainer ID in the entertainers table matches entertainer ID in the entertainer members table where the entertainer ID is in the selection of entertainer IDs from the entertainer styles table joined with the musical styles table on style ID in the entertainer styles table matches style ID in the musical styles table where the stylename is 'Jazz' and where the count of the members is greater than 3, grouped by entertainer stage name

Clean Up

Select ~~the~~ entertainer stage name
~~and the~~ count(*) ~~of members~~ as CountOfMembers
from ~~the~~ entertainers ~~table~~
inner join~~ed with the~~ entertainer members ~~table~~
on entertainers.entertainer ID ~~in the entertainers table~~
~~matches~~ = entertainer_members.entertainer ID
~~in the entertainer members table~~
where ~~the~~ entertainer ID ~~is~~ in ~~the~~
(select~~ion of~~ entertainer ID~~s~~
from ~~the~~ entertainer styles ~~table~~
inner join~~ed with the~~ musical styles ~~table~~
on entertainer_styles.style ID ~~in the entertainer styles table~~
~~matches~~ = musical_styles.style ID ~~in the musical styles table~~
where ~~the~~ style name ~~is~~ = 'Jazz')
and ~~where the~~ count(*) ~~of the members is greater than~~ > 3~~,~~
group~~ed~~ by entertainer stage name

SQL

```
SELECT Entertainers.EntStageName,
   COUNT(*) AS CountOfMembers
FROM Entertainers
INNER JOIN Entertainer_Members
ON Entertainers.EntertainerID =
   Entertainer_Members.EntertainerID
WHERE Entertainers.EntertainerID
IN
  (SELECT Entertainer_Styles.EntertainerID
   FROM Entertainer_Styles
   INNER JOIN Musical_Styles
   ON Entertainer_Styles.StyleID =
      Musical_Styles.StyleID
   WHERE Musical_Styles.StyleName = 'Jazz')
AND COUNT(*) > 3
GROUP BY Entertainers.EntStageName
```

What's wrong with this picture? The key is that any column you reference in a WHERE clause (remember Chapter 6?) *must* be a column in one of the tables defined in the FROM clause. Is COUNT(*) a column generated from the FROM clause? We don't think so! In fact, you can calculate COUNT for each group only after the rows are grouped.

Looks like we need a new clause after GROUP BY. Figure 14–2 shows the entire syntax for a SELECT statement, including the new HAVING clause.

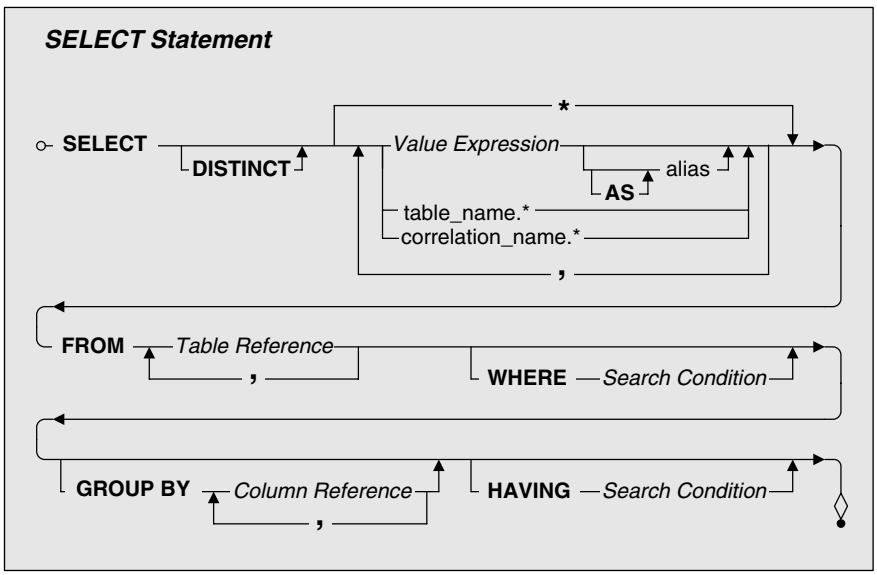


Figure 14–2 *The SELECT statement and all its clauses*

Because the HAVING clause acts on rows *after* they have been grouped, the SQL Standard defines some restrictions on the columns you reference in any predicate in the search condition. Note that when you do not have a GROUP BY clause, the HAVING clause operates on all rows returned by the FROM and WHERE clauses as though they are a single group.

The restrictions are the same as those for columns referenced in the SELECT clause of a grouped query. Any reference to a column in a predicate within the search condition of a HAVING clause either must name a column listed in the GROUP BY clause or must be enclosed within an aggregate function. This makes sense because any column comparisons must use something generated from the grouped rows—either a grouping value or an aggregate calculation across rows in each group.

Now that you know a bit about HAVING, let's solve the earlier problem in the correct way.

"Show me the entertainer groups that play in a jazz style and have more than three members."

Translation Select the entertainer stage name and the count of members from the entertainers table joined with the entertainer members table on entertainer ID in the entertainers table matches entertainer ID in the entertainer members table where the entertainer ID is in the selection of entertainer IDs from the entertainer styles table joined with the musical styles table on style ID in the entertainer styles table matches style ID in the musical styles table where the style name is 'Jazz,' grouped by entertainer stage name, and having the count of the members greater than 3

Clean Up Select ~~the~~ entertainer stage name
~~and the~~ count(*) ~~of members~~ as CountOfMembers
from ~~the~~ entertainers ~~table~~
inner join~~ed with the~~ entertainer members ~~table~~
on entertainers.entertainer ID ~~in the entertainers table matches~~
= entertainer_members.entertainer ID
~~in the entertainer members table~~
where ~~the~~ entertainer ID ~~is~~ in ~~the~~
(select~~ion of~~ entertainer ID~~s~~
from ~~the~~ entertainer styles ~~table~~
inner join~~ed with the~~ musical styles ~~table~~
on entertainer_styles.style ID
~~in the entertainer styles table matches~~
= musical_styles.style ID ~~in the musical styles table~~
where ~~the~~ style name ~~is~~ = 'Jazz')~~,~~
group~~ed~~ by entertainer stage name~~,~~
~~and~~ having ~~the~~ count(*) ~~of the members greater than~~ > 3

SQL

```
SELECT Entertainers.EntStageName,
   COUNT(*) AS CountOfMembers
FROM Entertainers
INNER JOIN Entertainer_Members
ON Entertainers.EntertainerID =
   Entertainer_Members.EntertainerID
WHERE Entertainers.EntertainerID
IN
  (SELECT Entertainer_Styles.EntertainerID
```

```
      FROM Entertainer_Styles
      INNER JOIN Musical_Styles
      ON Entertainer_Styles.StyleID =
         Musical_Styles.StyleID
      WHERE Musical_Styles.StyleName = 'Jazz')
GROUP BY Entertainers.EntStageName
HAVING COUNT(*) > 3
```

Although we also included the COUNT in the final output of the request, we didn't need to do that in order to ask for COUNT(*) in the HAVING clause. As long as any calculated value or column reference we use in HAVING can be derived from the grouped rows, we're OK. We saved this query in the Entertainment Agency sample database as CH14_Jazz_Entertainers_More_Than_3.

When You Filter Makes a Difference

You now know two ways to filter your final result set: WHERE and HAVING. You also know that there are certain limitations on the predicates you can use within a search condition in a HAVING clause. In some cases, however, you have the choice of placing a predicate in either clause. Let's take a look at the reasons for putting your filter in the WHERE clause instead of the HAVING clause.

Should You Filter in WHERE or in HAVING?

You learned in Chapter 6 about five major types of predicates you can build to filter the rows returned by the FROM clause of your request. These are comparison (=, <>, <, >, >=, <=), range (BETWEEN), set membership (IN), pattern match (LIKE), and Null (IS NULL). In Chapter 11, we expanded your horizons by showing you how to use a subquery as one of the arguments in comparison and set membership predicates, and we introduced you to two additional classes of predicates—quantified (ANY, SOME, ALL) and existence (EXISTS)—that require a subquery as one of the arguments.

Keep in mind that the search condition in a WHERE clause filters rows *before* your database system groups them. In general, when you want to ultimately group only a subset of rows, it's better to eliminate unwanted rows first in the WHERE clause. For example, let's assume you want to solve the following problem.

> *"Show me the states on the west coast of the United States where the total of the orders is greater than $1 million."*

Figure 14-3 shows the tables needed to solve this problem.

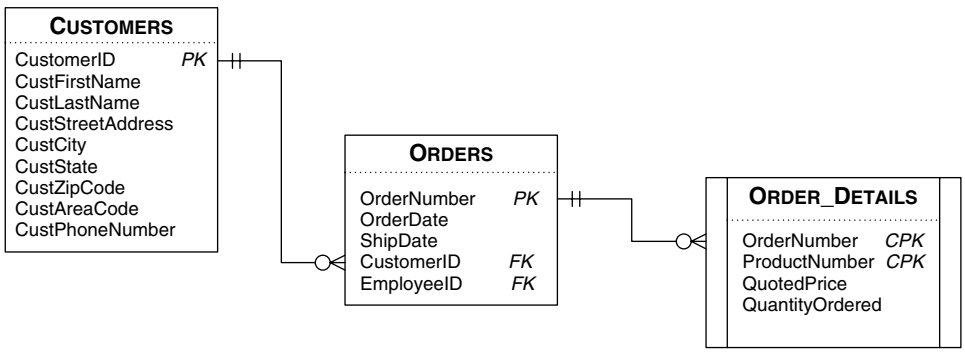


Figure 14–3 *The tables needed to sum all orders by state*

You could legitimately state the request in the following manner, placing the predicate on customer state into the HAVING clause.

```
SQL         SELECT Customers.CustState,
               SUM(Order_Details.QuantityOrdered *
               Order_Details.QuotedPrice) AS SumOfOrders
            FROM (Customers
               INNER JOIN Orders
               ON Customers.CustomerID = Orders.CustomerID)
            INNER JOIN Order_Details
            ON Orders.OrderNumber =
               Order_Details.OrderNumber
            GROUP BY Customers.CustState
            HAVING SUM(Order_Details.QuantityOrdered *
               Order_Details.QuotedPrice) > 1000000
            AND CustState IN ('WA', 'OR', 'CA')
```

Because you are grouping on the state column, you *can* construct a predicate on that column in the HAVING clause, but you might be asking your database system to do more work than necessary. As it turns out, the total of all orders for customers in the state of Texas also exceeds $1 million. If you place the filter on customer state in the HAVING clause as shown here, your database will calculate the total for all the rows in Texas as well, evaluate the first predicate in the HAVING clause and keep the result, and then finally throw it out when the Texas group isn't one you want.

If you want to calculate a result based on grouping by customer state but want only customers in Washington, Oregon, and California, it makes more sense to filter down to the rows in those three states using a WHERE clause before you

ask to GROUP BY state. If you don't do so, the FROM clause returns rows for all customers in all states and must do extra work to group rows you're not even going to need. Here's the better way to solve the problem.

Translation
Select customer state and the sum of quantity ordered times quoted price as SumOfOrders from the customers table joined with the orders table on customer ID in the customers table matches customer ID in the orders table, and then joined with the order details table on order number in the orders table matches order number in the order details table where customer state is in the list 'WA', 'OR', 'CA', grouped by customer state, and having the sum of the orders greater than $1 million

Clean Up
Select customer state, ~~and the~~
sum ~~of~~ (quantity ordered ~~times~~ * quoted price) as SumOfOrders
from ~~the~~ customers ~~table~~
inner join~~ed with the~~ orders ~~table~~
on customers.customer ID ~~in the customers table matches~~
= orders.customer ID ~~in the orders table,~~
~~and then~~ join~~ed with the~~ order details ~~table~~
on orders.order number ~~in the orders table matches~~
= order_details.order number ~~in the order details table~~
where customer state ~~is~~ in ~~the list~~ ('WA', 'OR', 'CA')~~,~~
group~~ed~~ by customer state~~, and~~
having ~~the~~ sum ~~of the orders~~ (quantity ordered * quoted price)
~~greater than~~ > ~~$1 million~~ 1000000

SQL

```
SELECT Customers.CustState,
   SUM(Order_Details.QuantityOrdered *
   Order_Details.QuotedPrice) AS SumOfOrders
FROM (Customers
   INNER JOIN Orders
   ON Customers.CustomerID = Orders.CustomerID)
INNER JOIN Order_Details
ON Orders.OrderNumber =
   Order_Details.OrderNumber
WHERE Customers.CustState IN ('WA', 'OR', 'CA')
GROUP BY Customers.CustState
HAVING SUM(Order_Details.QuantityOrdered *
   Order_Details.QuotedPrice) > 1000000
```

We saved this query in the sample database as CH14_West_Coast_Big_Order_States.

Avoiding the HAVING COUNT Trap

Many times you might want to know which categories of items have fewer than a certain number of members. For example, you might want to know which entertainment groups have two or fewer members, which recipes have two or fewer dairy ingredients, or which subjects have three or fewer full-time professors teaching. The trick here is you *also* want to know which categories have *zero* members.

Let's look at a request that illustrates the trap you can fall into.

> *"Show me the subject categories that have fewer than three full professors teaching that subject."*

Figure 14–4 shows the tables needed to solve this problem.

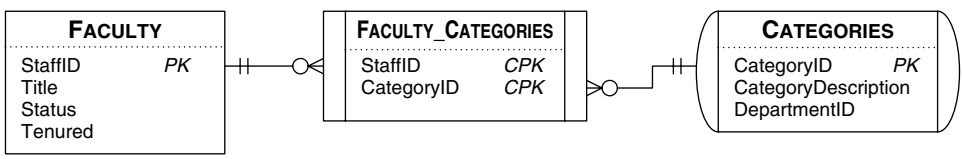


Figure 14–4 *The tables needed to find out which categories have fewer than three faculty teaching in that category*

Translation: Select category description and the count of staff ID as ProfCount from the categories table joined with the faculty categories table on category ID in the categories table matches category ID in the faculty categories table, and then joined with the faculty table on staff ID in the faculty table matches staff ID in the faculty categories table where title is 'Professor,' grouped by category description, and having the count of staff ID less than 3

Clean Up: Select category description
~~and the~~ count ~~of~~ (staff ID) as ProfCount
from ~~the~~ categories ~~table~~
inner join~~ed with the~~ faculty categories ~~table~~
on categories.category ID ~~in the categories table matches~~
= faculty_categories.category ID ~~in the faculty categories table, and then~~ inner join~~ed with the~~ faculty ~~table~~
on faculty.staff ID ~~in the faculty table matches~~

= faculty_categories.staff ID ~~in the faculty categories table~~
where title ~~is~~ = 'Professor~~,~~'
group~~ed~~ by category description~~,and~~
having ~~the~~ count ~~of~~ (staff ID) ~~less than~~ < 3

SQL

```
SELECT Categories.CategoryDescription,
   COUNT(Faculty_Categories.StaffID) AS
   ProfCount
FROM (Categories
INNER JOIN Faculty_Categories
ON Categories.CategoryID =
   Faculty_Categories.CategoryID)
INNER JOIN Faculty
ON Faculty.StaffID = Faculty_Categories.StaffID
WHERE Faculty.Title = 'Professor'
GROUP BY Categories.CategoryDescription
HAVING COUNT(Faculty_Categories.StaffID) < 3
```

Looks good, doesn't it? Below is the result set returned from this query.

CH14_Subjects_Fewer_3_Professors_WRONG

CategoryDescription	ProfCount
Accounting	1
Business	2
Computer Information Systems	1
Economics	1
Geography	1
History	1
Journalism	1
Math	1
Political Science	1

Do you notice that the result set lists *no* subject categories with zero professors? This happened because the COUNT function is counting only the rows that are left in the Faculty_Categories table after filtering for full professors. We threw away any potential zero rows with the WHERE clause!

Just to confirm our suspicions that some categories exist with no full professors, let's construct a query that will test our theory. Remember that the

COUNT aggregate function will return a zero if we ask it to count an empty set, and we can get an empty set if we force the request to consider how many rows exist for a specific subject category. We do this by forcing the query to look at the subject categories one at a time. We'll be counting category rows, not faculty subject rows. Consider the following SELECT statement.

```
SQL         SELECT COUNT(Faculty.StaffID)
            AS BiologyProfessors
            FROM (Faculty
            INNER JOIN Faculty_Categories
            ON Faculty.StaffID =
               Faculty_Categories.StaffID)
            INNER JOIN Categories
            ON Categories.CategoryID =
               Faculty_Categories.CategoryID
            WHERE Categories.CategoryDescription =
               'Biology'
            AND Faculty.Title = 'Professor'
```

BiologyProfessors
0

We saved this query as CH14_Count_Of_Biology_Professors in the sample database. As you can see, there really are no full professors in the School Scheduling sample database who teach biology. We asked the query to consider just one subject category. Because there are no rows that are both Professor and Biology, we get a legitimate empty set. The COUNT function, therefore, returns a zero.

Now that we know this, we can embed this request as a subquery in a WHERE clause that extracts a match on category ID from the outer query. This forces the request to consider the categories one at a time as it fetches the category descriptions one row at a time from the Categories table in the outer request. The SQL is as follows.

```
SQL         SELECT Categories.CategoryDescription,
              (SELECT COUNT(Faculty.StaffID)
               FROM (Faculty
               INNER JOIN Faculty_Categories
               ON Faculty.StaffID =
                  Faculty_Categories.StaffID)
```

```
   INNER JOIN Categories AS C2
   ON C2.CategoryID =
      Faculty_Categories.CategoryID
   WHERE C2.CategoryID = Categories.CategoryID
   AND Faculty.Title = 'Professor')
   AS ProfCount
FROM Categories
WHERE
  (SELECT COUNT(Faculty.StaffID)
   FROM (Faculty
   INNER JOIN Faculty_Categories
   ON Faculty.StaffID =
      Faculty_Categories.StaffID)
   INNER JOIN Categories AS C3
   ON C3.CategoryID =
      Faculty_Categories.CategoryID
   WHERE C3.CategoryID = Categories.CategoryID
   AND Faculty.Title = 'Professor') < 3
```

We saved this query as CH14_Subjects_Fewer_3_Professors_RIGHT in the sample database. Notice that we also included a copy of the subquery in the SELECT clause so that we can see the actual counts per category. This now works correctly because the subquery in the WHERE clause legitimately returns zero for a category that has no full professors. The correct result is below.

CH14_Subjects_Fewer_3_Professors_RIGHT

CategoryDescription	ProfCount
Accounting	1
Biology	0
Business	2
Chemistry	0
Computer Information Systems	1
Computer Science	0
Economics	1
Geography	1
History	1
Journalism	1
Math	1
Physics	0
Political Science	1
Psychology	0
French	0
German	0

As you can see, many subject categories actually have *no* full professors assigned to teach the subject. Although this final solution does not use HAVING at all, we include it to make you aware that HAVING isn't always the clear solution for this type of problem. Keep in mind that you can still use HAVING for many ". . . having fewer than . . ." problems. For example, if you want to see all customers who spent less than $500 last month, but you don't care about customers who bought nothing at all, then the HAVING solution works just fine (and will most likely execute faster). However, if you also need to see customers who bought nothing, you will have to use the non-HAVING technique we just showed you.

Uses for HAVING

At this point, you should have a good understanding of how to ask for subtotals across groups using aggregate functions and the GROUP BY clause and how to filter the grouped data using HAVING. The best way to give you an idea of the wide range of uses for HAVING is to list some problems you can solve with this new clause and then present a set of examples in the Sample Statements section.

"Show me each vendor and for each vendor the average of the number of days to deliver products, but display only the vendors whose average number of days to deliver is greater than the average number of delivery days for all vendors."

"Display for each product the product name and the total sales that are greater than the average of sales for all products in that category."

"List for each customer and order date the customer full name and the total cost of items ordered that is greater than $1,000."

"How many orders are for only one product?"

"Which agents booked more than $3,000 worth of business in December 2007?"

"Show me the entertainers who have more than two overlapped bookings."

"Show each agent name, the sum of the contract price for the engagements booked, and the agent's total commission for agents whose total commission is more than $1,000."

"Do any team captains have a raw score that is higher than any other member on the team?"

"Display for each bowler the bowler name and the average of the bowler's raw game scores for bowlers whose average is greater than 155."

"List the bowlers whose highest raw scores are at least 20 higher than their current averages."

"For completed classes, list by category and student the category name, the student name, and the student's average grade of all classes taken in that category for those students who have an average of 90 or better."

"Display by category the category name and the count of classes offered for those categories that have three or more classes."

"List each staff member and the count of classes each is scheduled to teach for those staff members who teach at least one but fewer than three classes."

"List the recipes that contain both beef and garlic."

"Sum the amount of salt by recipe class, and display those recipe classes that require more than three teaspoons."

"For what type of recipe do I have two or more recipes?"

Sample Statements

You now know the mechanics of constructing queries using a HAVING clause and have seen some of the types of requests you can answer. Let's take a look at a set of samples, all of which request that the information be grouped and then filtered on an aggregate value from the group. These examples come from each of the sample databases.

We've also included sample result sets that would be returned by these operations and placed them immediately after the SQL syntax line. The name that appears immediately above a result set is the name we gave each query in the sample data on the companion CD you'll find bound into the back of the book. We stored each query in the appropriate sample database (as indicated within the example), and we prefixed the names of the queries relevant to this chapter with "CH14." You can follow the instructions in the Introduction of this book to load the samples onto your computer and try them.

❖ **Note** Remember that all the column names and table names used in these examples are drawn from the sample database structures shown in Appendix B, Schema for the Sample Databases. To simplify the process, we have combined the Translation and Clean Up steps for all the examples. These samples assume you have thoroughly studied and understood the concepts covered in previous chapters, especially the chapters on JOINs and subqueries.

Sales Orders Database

"List for each customer and order date the customer's full name and the total cost of items ordered that is greater than $1,000."

Translation/ Clean Up

Select customer first name ~~and~~ || ' ' || customer last name
as CustFullName, order date, ~~and the~~
sum ~~of~~ (quoted price ~~times~~ * quantity ordered) as TotalCost
from ~~the~~ customers ~~table~~
inner join~~ed with the~~ orders ~~table~~
on customers.customer ID ~~in the customers table matches~~
= orders.customer ID ~~in the orders table,~~
~~and then~~ inner join~~ed with the~~ order details ~~table~~
on orders.order number ~~in the orders table matches~~
= order_details.order number ~~in the order details table,~~
group~~ed~~ by customer first name,
customer last name, ~~and~~ order date~~,~~
having ~~the~~ sum ~~of~~ (quoted price ~~times~~ * quantity ordered)
~~greater than~~ > 1000

SQL

```
SELECT Customers.CustFirstName || ' ' ||
   Customers.CustLastName AS CustFullName,
   Orders.OrderDate,
   SUM(Order_Details.QuotedPrice *
   Order_Details.QuantityOrdered) AS TotalCost
FROM (Customers
INNER JOIN Orders
ON Customers.CustomerID = Orders.CustomerID)
INNER JOIN Order_Details
ON Orders.OrderNumber =
   Order_Details.OrderNumber
GROUP BY Customers.CustFirstName,
   Customers.CustLastName, Orders.OrderDate
HAVING SUM(Order_Details.QuotedPrice *
   Order_Details.QuantityOrdered) > 1000
```

CH14_Order_Totals_By_Customer_And_Date_GT1000 (649 rows)

CustFullName	OrderDate	TotalCost
Alaina Hallmark	2007-09-02	$4,699.98
Alaina Hallmark	2007-09-14	$4,433.95
Alaina Hallmark	2007-09-21	$3,951.90
Alaina Hallmark	2007-09-22	$10,388.68
Alaina Hallmark	2007-09-30	$3,088.00
Alaina Hallmark	2007-10-12	$6,775.06
Alaina Hallmark	2007-10-22	$15,781.10
Alaina Hallmark	2007-10-30	$15,969.50
<< more rows here >>		

Entertainment Agency Database

"Which agents booked more than $3,000 worth of business in December 2007?"

Translation/ Clean Up

Select ~~the~~ agent first name, agent last name, ~~and the~~
sum ~~of~~ (contract price) as TotalBooked
from ~~the~~ agents ~~table~~
inner join~~ed with the~~ engagements ~~table~~
on agents.agent ID ~~in the agents table matches~~
= engagements.agent ID ~~in the engagements table~~
where ~~the~~ engagement start date ~~is~~
between ~~December 1, 2007,~~ '2007-12-01'
and ~~December 31, 2007,~~ '2007-12-31'~~,~~
group~~ed~~ by agent first name, ~~and~~ agent last name~~, and~~
having ~~the~~ sum ~~of~~ (contract price) ~~greater than~~ > 3000

SQL

```
SELECT Agents.AgtFirstName, Agents.AgtLastName,
   SUM(Engagements.ContractPrice)
   AS TotalBooked
FROM Agents
INNER JOIN Engagements
ON Agents.AgentID = Engagements.AgentID
WHERE Engagements.StartDate
BETWEEN '2007-12-01' AND '2007-12-31'
GROUP BY Agents.AgtFirstName, Agents.AgtLastName
HAVING SUM(Engagements.ContractPrice) > 3000
```

CH14_Agents_Book_Over_3000_12_2007 (2 rows)

AgtFirstName	AgtLastName	TotalBooked
Marianne	Weir	$6,650.00
William	Thompson	$3,340.00

School Scheduling Database

"For completed classes, list by category and student the category name, the student name, and the student's average grade of all classes taken in that category for those students who have an average higher than 90."

Translation/ Clean Up

Select category description, student first name, student last name,
~~and the average~~ avg(~~of~~ grade) as AvgOfGrade
from ~~the~~ categories ~~table~~
inner join~~ed with the~~ subjects ~~table~~
on categories.category ID ~~in the categories table matches~~
= subjects.category ID ~~in the subjects table,~~
~~then~~ inner join~~ed with the~~ classes ~~table~~
on subjects.subject ID ~~in the subjects table matches~~
= classes.subject ID ~~in the classes table,~~
~~then~~ inner join~~ed with the~~ student schedules ~~table~~
on classes.class ID ~~in the classes table matches~~
= student_schedules.class ID ~~in the student schedules table,~~
~~then~~ inner join~~ed with the~~ student class status ~~table~~
on student_class_status.class status ~~in the student class status table matches~~
= student_schedules.class status ~~in the student schedules table,~~
~~and finally~~ inner join~~ed with the~~ students ~~table~~
on students.student ID ~~in the students table matches~~
= student_schedules.student ID ~~in the student schedules table~~
where class status description ~~is~~ = 'Completed~~,~~'
group~~ed~~ by category description, student first name, ~~and~~
student last name~~,~~
~~and~~ having ~~the average~~ avg(~~of~~ grade) ~~greater than~~ > 90

SQL

```
SELECT Categories.CategoryDescription,
   Students.StudFirstName,
   Students.StudLastName,
   AVG(Student_Schedules.Grade) AS AvgOfGrade
FROM ((((Categories
INNER JOIN Subjects
ON Categories.CategoryID = Subjects.CategoryID)
INNER JOIN Classes
ON Subjects.SubjectID = Classes.SubjectID)
INNER JOIN Student_Schedules
ON Classes.ClassID = Student_Schedules.ClassID)
INNER JOIN Student_Class_Status
ON Student_Class_Status.ClassStatus =
   Student_Schedules.ClassStatus)
INNER JOIN Students
ON Students.StudentID =
```

```
   Student_Schedules.StudentID
WHERE Student_Class_Status.ClassStatusDescription =
   'Completed'
GROUP BY Categories.CategoryDescription,
   Students.StudFirstName,
   Students.StudLastName
HAVING AVG(Student_Schedules.Grade) > 90
```

CH14_A_Students (13 rows)

CategoryDescription	StudFirstName	StudLastName	AvgOfGrade
Accounting	Betsey	Stadick	90.67
Accounting	Elizabeth	Hallmark	90.24
Accounting	Michael	Viescas	90.01
Art	Kendra	Bonnicksen	90.63
Art	Sarah	Thompson	91.40
English	Brannon	Jones	93.86
English	Elizabeth	Hallmark	92.90
English	John	Kennedy	93.70
English	Sarah	Thompson	97.39
Music	Brannon	Jones	93.26
Music	Karen	Smith	92.08
Music	Kerry	Patterson	93.28
Music	Marianne	Wier	98.26

"List each staff member and the count of classes each is scheduled to teach for those staff members who teach at least one but fewer than three classes."

❖ **Note** We avoided the HAVING COUNT zero problem by specifically stating that we want staff members who teach at least one class.

Translation/ Clean Up: Select staff first name, staff last name, ~~and the~~ count ~~of classes~~ (*) as ClassCount from ~~the~~ staff ~~table~~ inner join~~ed with the~~ faculty classes ~~table~~ on staff.staff ID ~~in the staff table matches~~ = faculty_classes.staff ID ~~in the faculty classes table,~~ group~~ed~~ by staff first name, ~~and~~ staff last name~~, and~~ having ~~the~~ count ~~of classes~~ (*) ~~less than~~ < 3

SQL:

```
SELECT Staff.StfFirstName, Staff.StfLastName,
   COUNT(*) AS ClassCount
FROM Staff
INNER JOIN Faculty_Classes
ON Staff.StaffID = Faculty_Classes.StaffID
GROUP BY Staff.StfFirstName, Staff.StfLastName
HAVING COUNT(*) < 3
```

CH14_Staff_Class_Count_1_To_3 (8 rows)

StfFirstName	StfLastName	ClassCount
Jim	Wilson	2
Joyce	Bonnicksen	2
Katherine	Ehrlich	2
Kirk	DeGrasse	2
Luke	Patterson	1
Mariya	Sergienko	2
Peter	Brehm	2
Suzanne	Viescas	2

Bowling League Database

"List the bowlers whose highest raw scores are more than 20 pins higher than their current averages."

Translation/ Clean Up

Select bowler first name, bowler last name,
~~the average~~ avg(raw score) as CurrentAverage,
~~and the~~ max~~imum~~ (raw score) as HighGame
from ~~the~~ bowlers ~~table~~
inner join~~ed with the~~ bowler scores ~~table~~
on bowlers.bowler ID ~~in the bowlers table matches~~
= bowler_scores.bowler ID ~~in the bowler scores table,~~
group~~ed~~ by bowler first name, ~~and~~ bowler last name~~, and~~
having ~~the~~ max~~imum~~ (raw score)
~~greater than~~ > ~~the average~~ avg(raw score) ~~plus~~ + 20

SQL

```
SELECT Bowlers.BowlerFirstName,
       Bowlers.BowlerLastName,
   AVG(Bowler_Scores.RawScore) AS CurrentAverage,
   MAX(Bowler_Scores.RawScore) AS HighGame
FROM Bowlers INNER JOIN Bowler_Scores
ON Bowlers.BowlerID = Bowler_Scores.BowlerID
GROUP BY Bowlers.BowlerFirstName,
         Bowlers.BowlerLastName
HAVING MAX(Bowler_Scores.RawScore) >
   (AVG(Bowler_Scores.RawScore) + 20)
```

CH14_Bowlers_Big_High_Score (15 rows)

BowlerFirstName	BowlerLastName	CurrentAverage	HighGame
Alaina	Hallmark	158	180
Angel	Kennedy	163	194
Caleb	Viescas	164	193
David	Fournier	157	178
David	Viescas	168	195
Gary	Hallmark	157	179
John	Kennedy	166	191
John	Viescas	168	193
<< more rows here >>			

Recipes Database

"List the recipes that contain both beef and garlic."

Translation/ Clean Up

Select recipe title from ~~the~~ recipes ~~table~~
where ~~the~~ recipe ID ~~is~~ in ~~the~~
(select~~ion of~~ recipe ID
from ~~the~~ ingredients ~~table~~
inner join~~ed with the~~ recipe ingredients ~~table~~
on recipe_ingredients.ingredient ID
~~in the recipe ingredients table~~
~~matches~~ = ingredients.ingredient ID ~~in the ingredients table~~
where ~~the~~ ingredient name ~~is~~ = 'Beef'
or ~~the~~ ingredient name ~~is~~ = 'Garlic~~,~~'
group~~ed~~ by recipe ID ~~and~~
having ~~the~~ count ~~of the values in~~ (recipe ID) ~~equal to~~ = 2)

SQL

```
SELECT Recipes.RecipeTitle
FROM Recipes
WHERE Recipes.RecipeID
IN (SELECT Recipe_Ingredients.RecipeID
   FROM Ingredients
   INNER JOIN Recipe_Ingredients
   ON Ingredients.IngredientID =
      Recipe_Ingredients.IngredientID
   WHERE Ingredients.IngredientName = 'Beef'
   OR Ingredients.IngredientName = 'Garlic'
   GROUP BY Recipe_Ingredients.RecipeID
   HAVING COUNT(Recipe_Ingredients.RecipeID) = 2)
```

CH14_Recipes_Beef_And_Garlic (1 row)

RecipeTitle
Roast Beef

❖ **Note** This illustrates a creative use of GROUP BY and HAVING in a subquery to find recipes that have *both* ingredients. When a recipe has neither of the ingredients, the recipe won't appear in the subquery. When a recipe has only one of the ingredients, the count will be 1, so the row will be eliminated. Only when a recipe has both will the COUNT be 2. Be careful, though. If a particular recipe calls for both minced and whole garlic but no beef, this technique won't work! You will get a COUNT of 2 for the two

garlic entries, so the recipe will be selected even though it has no beef. If you wonder why we used an OR operator when we want both beef and garlic, be sure to review the Using OR topic in the Using Multiple Conditions section in Chapter 6. We showed you an alternative way to solve this problem in Chapter 8.

SUMMARY

We started the chapter with a discussion about focusing the groups you form by using the HAVING clause to filter out groups based on aggregate calculations. We introduced the syntax of this final clause for a SELECT statement and explained a simple example.

Next we showed an example of when to use the WHERE clause rather than the HAVING clause to filter rows. We explained that when you have a choice, you're better off placing your filter in the WHERE clause. Before you got too comfortable with HAVING, we showed you a common trap to avoid when counting groups that might contain a zero result. We also showed you an alternative way to solve this type of problem.

Finally, we summarized why the HAVING clause is useful and gave you a sample list of problems you can solve using HAVING. The rest of the chapter provided examples of how to build requests that require the HAVING clause.

The following section presents a number of requests that you can work out on your own.

Problems for You to Solve

Below, we show you the request statement and the name of the solution query in the sample databases. If you want some practice, you can work out the SQL you need for each request and then check your answer with the query we saved in the samples. Don't worry if your syntax doesn't exactly match the syntax of the queries we saved—as long as your result set is the same.

Sales Orders Database

1. *"Show me each vendor and the average by vendor of the number of days to deliver products that is greater than the average delivery days for all vendors."* (Hint: You need a subquery to fetch the average delivery time for all vendors.)

You can find the solution in CH14_Vendor_Avg_Delivery_GT_Overall_Avg (5 rows).

2. *"Display for each product the product name and the total sales that is greater than the average of sales for all products in that category."*
   (Hint: To calculate the comparison value, you must first SUM the sales for each product within a category and then AVG those sums by category.)
   You can find the solution in CH14_Sales_By_Product_GT_Category_Avg (13 rows).
3. *"How many orders are for only one product?"*
   (Hint: You need to use an inner query in the FROM clause that lists the order numbers for orders having only one row and then COUNT those rows in the outer SELECT clause.)
   You can find the solution in CH14_Single_Item_Order_Count (1 row).

Entertainment Agency Database

1. *"Show me the entertainers who have more than two overlapped bookings."*
   (Hint: Use a subquery to find those entertainers with overlapped bookings HAVING a COUNT greater than 2.)
   You can find the solution in CH14_Entertainers_MoreThan_2_Overlap (1 row).
2. *"Show each agent's name, the sum of the contract price for the engagements booked, and the agent's total commission for agents whose total commission is more than $1,000."*
   (Hint: Use the similar problem from Chapter 13 and add a HAVING clause.)
   You can find the solution in CH14_Agent_Sales_Big_Commissions (4 rows).

School Scheduling Database

1. *"Display by category the category name and the count of classes offered for those categories that have three or more classes."*
   (Hint: JOIN categories to subjects and then to classes. COUNT the rows and add a HAVING clause to get the final result.)
   You can find the solution in CH14_Category_Class_Count_3_Or_More (11 rows).
2. *"List each staff member and the count of classes each is scheduled to teach for those staff members who teach fewer than three classes."*
   (Hint: This is a HAVING COUNT zero trap! Use subqueries instead.)
   You can find the solution in CH14_Staff_Teaching_LessThan_3 (12 rows).
3. *"Count the classes taught by all staff members."*
   (Hint: This really isn't a HAVING problem, but you might be tempted to solve it incorrectly using a GROUP BY.)
   You can find the correct solution in CH14_Staff_Class_Count_Subquery (27 rows).
   The incorrect solution is in CH14_Staff_Class_Count_GROUPED (23 rows).

Bowling League Database

1. *"Do any team captains have a raw score that is higher than any other member on the team?"*
   (Hint: You find out the top raw score for captains by JOINing teams to bowlers on captain ID and then to bowler scores. Use a HAVING clause to compare the MAX value for all other members from a subquery.)
   You can find the solution in CH14_Captains_Who_Are_Hotshots (0 rows).
   (There are no captains who bowl better than their teammates!)
2. *"Display for each bowler the bowler name and the average of the bowler's raw game scores for bowlers whose average is greater than 155."*
   (Hint: You need a simple HAVING clause comparing the AVG to a numeric literal.)
   You can find the solution in CH14_Good_Bowlers (17 rows).
3. *"List the last name and first name of every bowler whose average raw score is greater than or equal to the overall average score."*
   (Hint: We showed you how to solve this in Chapter 12 in the Sample Statements section with a subquery in a WHERE clause. Now solve it using HAVING!)
   You can find the solution in CH14_Better_Than_Overall_Average_HAVING (17 rows).

Recipes Database

1. *"Sum the amount of salt by recipe class, and display those recipe classes that require more than 3 teaspoons."*
   (Hint: This requires a complex JOIN of five tables to filter out salt and teaspoon, SUM the result, and then eliminate recipe classes that use more than 3 teaspoons.)
   You can find the solution in CH14_Recipe_Classes_Lots_Of_Salt (1 row).
2. *"For what class of recipe do I have two or more recipes?"*
   (Hint: JOIN recipe classes with recipes, count the result, and keep the ones with two or more with a HAVING clause.)
   You can find the solution in CH14_Recipe_Classes_Two_Or_More (4 rows).

Part V

Modifying Sets of Data

This page intentionally left blank

15

Updating Sets of Data

"It is change, continuing change, inevitable change, that is the dominant factor in society today."
—Isaac Asimov

Topics Covered in This Chapter

What Is an UPDATE?

The UPDATE Statement

Uses for UPDATE

Sample Statements

Summary

Problems for You to Solve

As you learned in Part II, SQL Basics; Part III, Working with Multiple Tables; and Part IV, Summarizing and Grouping Data, using the SELECT statement to fetch data from your tables can be both fun and challenging. (OK, so maybe some of it is a lot more challenging than fun!) If all you need to do is answer questions, then you don't need this last part of our book. However, most real-world applications not only answer complex questions but also allow the user to change, add, or delete data. In addition to defining the SELECT statement that you've been learning about to retrieve data, the SQL Standard also defines three statements that allow you to modify your data. In this chapter, you'll learn about the first of those statements—UPDATE.

What Is an UPDATE?

The SELECT statement lets you retrieve sets of data from your tables. The UPDATE statement also works with sets of data, but you can use it to *change*

the values in one or more columns and in one or more rows. By now, you should also be very familiar with expressions. To change a value in a column, you simply assign an expression to the column.

But you must be careful because UPDATE is very powerful. Most of the time you'll want to update only one or a few rows. If you're not careful, you can end up changing thousands of rows. To avoid this problem, we'll show you a technique for testing your statement first.

❖ **Note** You can find all the sample statements and solutions in the "modify" version of the respective sample databases—SalesOrdersModify, EntertainmentAgencyModify, SchoolSchedulingModify, and BowlingLeagueModify.

The UPDATE Statement

The UPDATE statement is actually much simpler than the SELECT statement that you have been learning about in the previous chapters. The UPDATE statement has only three clauses: UPDATE, SET, and an optional WHERE clause, as shown in Figure 15–1.

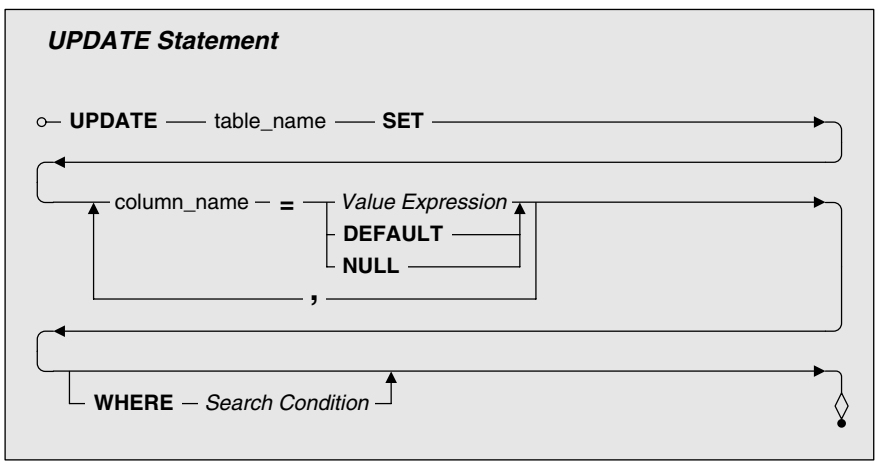


Figure 15–1 *The syntax diagram of the UPDATE statement*

After the UPDATE keyword, you specify the name of the table that you want to update. The SET keyword begins one or more clauses that assign a new value to a column in the table. You must include at least one assignment

clause, and you can include as many assignment clauses as you need to change the value of multiple columns in each row. Use the optional WHERE clause to restrict the rows that are to be updated in the target table.

Using a Simple UPDATE Expression

Let's look at an example using a simple assignment of an expression to the column you want to update.

> ❖ **Note** Throughout this chapter, we use the "Request/Translation/Clean Up/SQL" technique introduced in Chapter 4, Creating a Simple Query.

"Increase the retail price of all products by 10 percent."

Ah, this is somewhat tricky. You'll find it tough to directly translate your original request into SQL-like English because you don't usually state the clauses in the same order required by the UPDATE statement. Take a close look at your request and figure out (a) the name of the target table and (b) the names of the columns you need to update. Restate your request in that order, and then proceed with the translation, like this:

"Change products by increasing the retail price by 10 percent."

Translation	Update the products table by setting the retail price equal to the retail price plus 10 percent of the price
Clean Up	Update ~~the~~ products ~~table by~~ set~~ting the~~ retail price ~~equal to~~ = ~~the~~ retail price ~~plus~~ + (.1~~0 percent of the~~ * retail price)
SQL	`UPDATE Products` `SET Price = Price + (0.1 * Price)`

Notice that you cannot say SET Price + 10 percent. You must state the column to be updated to the left of the equals sign and then create an expression to calculate the new value you want. If the new value involves using the old or current value of the column, then you must reference the column name as needed to the right of the equals sign. One rule that's very clear in the SQL Standard is that your database system must evaluate all the assignment expressions *before* it updates any rows. So your database will resolve the two references to the

Price column to the right of the equals sign by fetching the value of the Price column before it makes any changes.

You'll find this sort of assignment statement common in any programming language. Although it might appear to you that you're assigning the value of a column to itself, you're really grabbing the value before it changes, adding 10 percent of the value, and then assigning the result to the column to update it to a new value.

Updating Selected Rows

Are you always going to want to update all rows in a table? Probably not. To limit the rows changed by your UPDATE statement, you need to add a WHERE clause. Let's consider another problem.

> *"My clothing supplier just announced a price increase of 4 percent. Update the price of the clothing products and add 4 percent."*

Let's restate that.

> *"Modify products by increasing the retail price by 4 percent for products that are clothing (category 3)."*

Translation	Update the products table by setting the retail price equal to retail price times 1.04 for all products in category 3
Clean Up	Update ~~the~~ products ~~table by~~ set~~ting the~~ retail price ~~equal to~~ = retail price ~~times~~ * 1.04 ~~for all~~ where ~~products in~~ category ID = 3

SQL

```
UPDATE Products
SET RetailPrice = RetailPrice * 1.04
WHERE CategoryID = 3
```

❖ **Note** We simplified the calculation in the query by multiplying the original value by 1.04 rather than adding the original value to 0.04 times the original value. The result is mathematically the same and might actually execute faster because one mathematical operation (price times 1.04) is more efficient than two (price plus price times .04).

After tackling subqueries in Chapter 11, this was easy, right? Just wait—you'll use subqueries extensively in your WHERE clauses, and we'll cover that later in this chapter.

Safety First: Ensure You're Updating the Correct Rows

Even for simple UPDATE queries, we strongly recommend that you verify that you're going to be updating the correct rows. How do you do that? As we mentioned, most of the time you'll add a WHERE clause to select a subset of rows to update. Why not build a SELECT query first to return the rows that you intend to update? Continuing with our example, let's ask the database to return a column that lets us ensure that we have the correct rows, the value we want to update, and the expression we intend to assign to the column we're changing.

"List the product name, retail price, and retail price plus 4 percent from the products table for the products in category 3."

Translation: Select product name, retail price, and retail price times 1.04 from the products table for products in category ID 3

Clean Up: Select product name, retail price, ~~and~~ retail price ~~times~~ * 1.04 from ~~the~~ products ~~table~~ ~~for~~ where ~~products in~~ category ID = 3

SQL:

```
SELECT ProductName, RetailPrice,
   RetailPrice * 1.04 As NewPrice
FROM Products
WHERE CategoryID = 3
```

Figure 15–2 shows the result.

ProductName	RetailPrice	NewPrice
Ultra-Pro Rain Jacket	$85.00	$88.40
StaDry Cycling Pants	$69.00	$71.76
Kool-Breeze Rocket Top Jersey	$32.00	$33.28
Wonder Wool Cycle Socks	$19.00	$19.76

Figure 15–2 *Verifying the rows you want to update*

Note that we included the product name so we can see exactly what we want to update. If this is the result we want, we can transform the SELECT statement into the correct UPDATE statement by removing elements we don't need and swapping the FROM and SELECT clauses. Figure 15–3 shows how to transform this SELECT statement into the correct UPDATE syntax.

Simply cross out the words you don't need, move the table name to the UPDATE clause, move the target field and expression to the SET clause separated by an equals sign, copy your WHERE clause, and you're done.

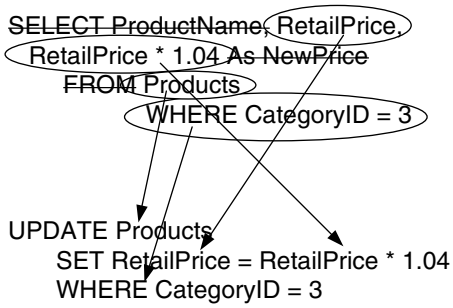


Figure 15–3 *Converting a SELECT query into an UPDATE statement*

A Brief Aside: Transactions

Before we get too much further into changing data, you need to know about an important feature available in SQL. The SQL Standard defines something called a *transaction* that you can use to protect a series of changes you're making to the data in your tables. You can think of a transaction in SQL just like a transaction you might make online or at a store to buy something. You initiate the transaction when you send in your order. Paying for the item you ordered is part of the transaction. The transaction is completed when you receive and accept the merchandise. But if the merchandise doesn't arrive, you might apply for a refund. Or if the merchandise is unsatisfactory, you return it and ask for your money back.

The SQL Standard provides three statements that mimic this scenario. You can use START TRANSACTION before you begin your changes to indicate that you want to protect and verify the changes you're about to make. Think of this as sending in your order. You make the changes to your database—register the payment and register the receipt. If everything completes satisfactorily, you can use COMMIT to make the changes permanent. If something went wrong (the payment or receipt update failed), you can use ROLLBACK to restore the data as it was before you started the transaction.

This buying and selling example might seem silly, but transactions are a very powerful feature of SQL, especially when you need to make changes to multiple rows or to rows in several tables. Using a transaction ensures that either all changes are successful or none are. You don't want to register the payment without receipt of the goods, and you don't want to mark the goods received without receiving the payment. Note that this applies to changing your data not only with the UPDATE statement described in this chapter but also with INSERT and DELETE, which are described in the next two chapters.

Not all database systems implement transactions, and the syntax to use transaction processing varies slightly depending on the particular database system. Some database systems allow you to nest transactions inside each other so that you can establish multiple commit points. Some end-user database systems, such as Microsoft Office Access, start a transaction for you behind the scenes every time you run a query that changes your data. If you've used Microsoft Access, you know that it prompts you with a message indicating how many rows will be changed and whether any will fail—and you can either accept the changes or cancel them (ROLLBACK). As always, consult your database documentation for details.

Updating Multiple Columns

As implied by the diagram of the UPDATE statement in Figure 15–1 (see page 502), you can specify more than one column to change by including additional assignment statements separated by columns. Keep in mind that your database applies all the changes you specify to every row returned as a result of evaluating your WHERE clause. Let's take a look at an update you might want to perform in the School Scheduling database.

"Modify classes by changing the classroom to 1635 and the schedule dates from Monday-Wednesday-Friday to Tuesday-Thursday-Saturday for all drawing classes (subject ID 13)."

Translation
: Update classes and set classroom ID to 1635, Monday schedule to false, Wednesday schedule to false, Friday schedule to false, Tuesday schedule to true, Thursday schedule to true, and Saturday schedule to true for all classes that are subject ID 13

Clean Up
: Update classes ~~and~~
set classroom ID ~~to~~ = 1635, Monday schedule ~~to~~ = false,
Wednesday schedule ~~to~~ = false, Friday schedule ~~to~~ = false,
Tuesday schedule ~~to~~ = true, Thursday schedule ~~to~~ = true,
~~and~~ Saturday schedule ~~to~~ = true
~~for all classes that are~~ where subject ID = 13

SQL
:
```
UPDATE Classes
SET ClassRoomID = 1635, MondaySchedule = 0,
   WednesdaySchedule = 0, FridaySchedule = 0,
   TuesdaySchedule = 1, ThursdaySchedule = 1,
   SaturdaySchedule = 1
WHERE SubjectID = 13
```

❖ **Note** Remember that most database systems use the value 0 for false and the value 1 or –1 for true. Check your database documentation for details.

Perhaps you want to make doubly sure that you're changing only the classes scheduled on Monday-Wednesday-Friday. To do that, add criteria to your WHERE clause like this.

```
SQL         UPDATE Classes
            SET ClassRoomID = 1635, MondaySchedule = 0,
               WednesdaySchedule = 0, FridaySchedule = 0,
               TuesdaySchedule = 1, ThursdaySchedule = 1,
               SaturdaySchedule = 1
            WHERE SubjectID = 13
               AND MondaySchedule = 1
               AND WednesdaySchedule = 1
               AND FridaySchedule = 1
```

Notice that you're filtering for the value you expect to find in the field *before* your UPDATE statement changes the value. With this modified query, you're finding all rows for SubjectID 13 that have a true (1) value in the Monday, Wednesday, and Friday schedule fields. For each row that matches these criteria, your UPDATE statement will change the ClassRoomID and the schedule fields. If you try to run this query a second time, you should find that your database updates no rows because you eliminated all rows that qualify by changing the field values the first time you ran the query.

Using a Subquery to Filter Rows

In the examples in previous sections, we've updated the products in category 3 and the classes in subject 13. In the real world, code values like this don't have much meaning. You'd probably much rather say "clothing products" or "drawing classes." In a SELECT query, you can add the related tables to your FROM clause with JOIN specifications and then display the more meaningful value from the related table. As always, you must be familiar with your table relationships to make this connection. Figure 15–4 shows the tables we need for our example.

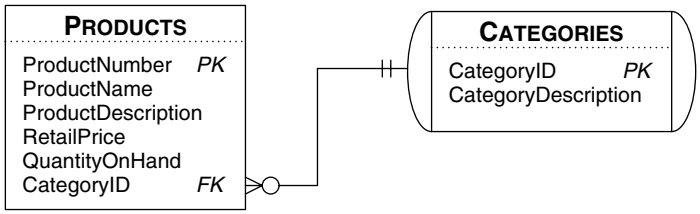


Figure 15–4 *The tables needed to relate category descriptions to products*

Let's look again at the verification query we built to check our update of products, but this time, let's add the Categories table.

```
SQL         SELECT ProductName, RetailPrice,
               RetailPrice * 1.04 As NewPrice
            FROM Products
            INNER JOIN Categories
            ON Products.CategoryID = Categories.CategoryID
            WHERE Categories.CategoryDescription = 'Clothing'
```

Filtering on the value Clothing makes a lot more sense than selecting the category ID value 3. However, notice that the diagram of the UPDATE statement in Figure 15–1 shows that we can supply only a table name following the UPDATE keyword. We cannot specify the INNER JOIN needed to include the Categories table so that we can filter on the more meaningful value. So what's the solution?

Remember from Chapter 11 that we can create a filter in a WHERE clause to test a value fetched from a related table. Let's solve the price update problem again using a subquery so that we can apply a more meaningful filter value.

"Modify products by increasing the retail price by 4 percent for products that are clothing."

Translation Update the products table by setting the retail price equal to retail price times 1.04 for the products whose category ID is equal to the selection of the category ID from the categories table where the category description is clothing

Clean Up Update ~~the~~ products ~~table by~~
set~~ting the~~ retail price ~~equal to~~ = retail price ~~times~~ * 1.04
~~for the products whose~~ where category ID ~~is equal to~~ =
~~the~~ (select~~ion of the~~ category ID
from ~~the~~ categories ~~table~~
where ~~the~~ category description ~~is~~ = 'Clothing')

```
SQL         UPDATE Products
            SET RetailPrice = RetailPrice * 1.04
            WHERE CategoryID =
               (SELECT CategoryID
               FROM Categories
               WHERE CategoryDescription = 'Clothing')
```

That's not as straightforward as a simple WHERE clause on a column from a joined table, but it gets the job done.

❖ **Caution** Notice that we used an equals comparison for the CategoryID column in the Products table and the value returned by the subquery. As we noted in Chapter 11, if you want to use an equals comparison in a predicate with a subquery, the subquery must return only one value. If more than one row in the Categories table had the value Clothing in the category description field, this query would fail. However, in our example, we're reasonably certain that filtering for Clothing will return only one value for CategoryID. Whenever you're not sure that a subquery will return only one value, you should use the IN predicate rather than the "equal to" operator.

Let's solve the problem of updating classes by using the same technique. We want to use the subject code or subject name from the Subjects table rather than the numeric and meaningless subject ID. Figure 15–5 shows the tables involved.

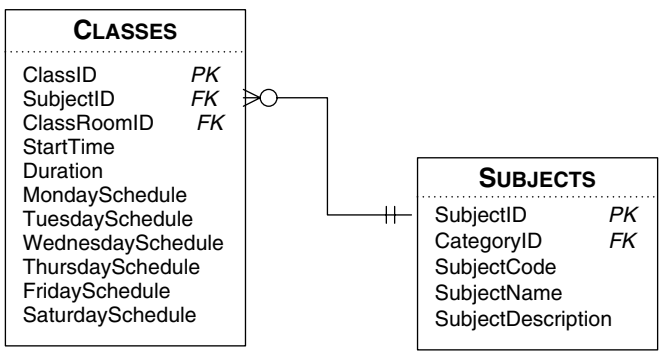


Figure 15–5 *The tables needed to relate subject names to classes*

Let's solve the update problem again by using a subquery filter.

"Modify classes by changing the classroom to 1635 and the schedule dates from Monday-Wednesday-Friday to Tuesday-Thursday-Saturday for all drawing classes."

Translation
Update classes and set classroom ID to 1635, Monday schedule to false, Wednesday schedule to false, Friday schedule to false, Tuesday schedule to true, Thursday schedule to true, and Saturday schedule to true for all classes whose subject ID is in the selection of subject IDs from the subjects table where subject name is 'Drawing'

Clean Up
Update classes ~~and~~
set classroom ID ~~to~~ = 1635, Monday schedule ~~to~~ = false,
Wednesday schedule ~~to~~ = false, Friday schedule ~~to~~ = false,
Tuesday schedule ~~to~~ = true, Thursday schedule ~~to~~ = true,
~~and~~ Saturday schedule ~~to~~ = true
~~for all classes whose~~ where subject ID ~~is~~ in
~~the~~ (select~~ion of~~ subject ID~~s~~
from ~~the~~ subjects ~~table~~
where subject name ~~is~~ = 'Drawing')

SQL

```
UPDATE Classes
SET ClassRoomID = 1635, MondaySchedule = 0,
   WednesdaySchedule = 0, FridaySchedule = 0,
   TuesdaySchedule = 1, ThursdaySchedule = 1,
   SaturdaySchedule = 1
WHERE SubjectID IN
   (SELECT SubjectID
   FROM Subjects
   WHERE SubjectName = 'Drawing')
```

Notice that even though we're fairly certain that only one subject ID has a subject name equal to Drawing, we decided to play it safe and use the IN predicate.

Some Database Systems Allow a JOIN in the UPDATE Clause

Several database systems, most notably the ones from Microsoft (Microsoft Access and Microsoft SQL Server), allow you to specify a joined table in the FROM clause of an UPDATE query. The restriction is that the JOIN must be

(*continued*)

from the primary key in one table to the foreign key in another table so that the database system can figure out which specific row or rows you intend to update. This allows you to avoid a subquery in the WHERE clause when you want to filter rows based on a value in a related table.

If your database system allows this, you can solve the problem of modifying the information on drawing classes as follows:

```
SQL         UPDATE Classes
            INNER JOIN Subject
            ON Classes.SubjectID = Subjects.SubjectID
            SET ClassRoomID = 1635, MondaySchedule = 0,
               WednesdaySchedule = 0, FridaySchedule = 0,
               TuesdaySchedule = 1, ThursdaySchedule = 1,
               SaturdaySchedule = 1
            WHERE Subjects.SubjectName = 'Drawing'
```

As you can see, this avoids having to use a subquery to filter the rows. In some ways, this syntax is also easier to understand. You can also use this syntax to join a related table that supplies one of the values in your update calculation rather than use a subquery in the SET clause. Be sure to check the documentation for your database system to see if this feature is supported. You'll note that we've used this technique to solve some of the sample queries in the Microsoft Access versions of the sample databases.

By the way, the SQL Standard allows the target table to be a view, which could imply a joined table. However, the Standard specifies that the rules for updating a view are defined by the implementation, which allows database system vendors to either always require a simple table name or otherwise restrict what you can do using a view or joined table. As always, check your database documentation for details.

As you might imagine, you can make the subquery as complex as necessary to allow you to properly filter the target table. For example, if you want to change the start time for all classes taught by one professor, you need to join the Faculty_Classes and Staff tables in the FROM clause of the subquery. Figure 15–6 shows the tables involved.

Let's say you want to change the start time of all classes taught by Kathryn Patterson to 2:00 P.M. (You probably wouldn't want to do this because you might end up with multiple classes starting at the same time, but this makes an interesting example.) Your solution might look like this:

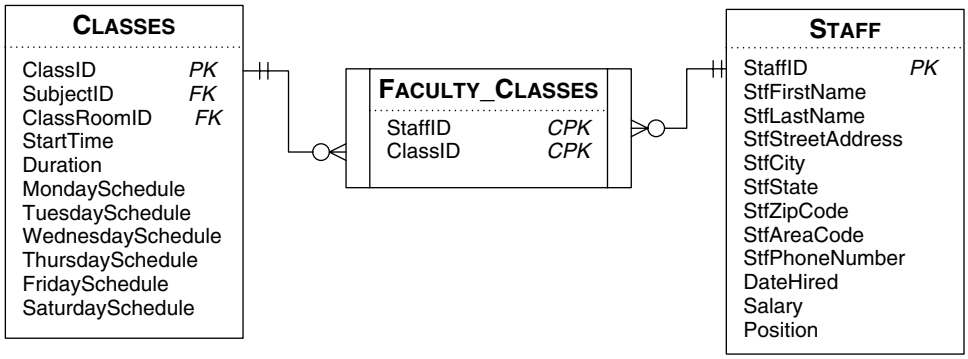


Figure 15–6 *The tables needed to relate staff names to classes*

"Change the classes table by setting the start time to 2:00 P.M. for all classes taught by Kathryn Patterson."

Translation Update the classes table by setting the start time to 2:00 P.M. for all classes whose class ID is in the selection of class IDs of faculty classes joined with staff on staff ID in the faculty classes table matches staff ID in the staff table where the staff first name is 'Kathryn' and the staff last name is 'Patterson'

Clean Up Update ~~the~~ classes ~~table by~~
setting ~~the~~ start time ~~to~~ = ~~2:00 P.M.~~ '14:00:00'
~~for all classes whose~~ where class ID ~~is~~ in
~~the~~ (select~~ion of~~ class ID~~s of~~ from faculty classes
inner join~~ed with~~ staff
on faculty_classes.staff ID ~~in the faculty classes table matches~~
= staff.staff ID ~~in the staff table~~
where ~~the~~ staff first name ~~is~~ = 'Kathryn'
and ~~the~~ staff last name ~~is~~ = 'Patterson')

SQL

```
UPDATE Classes
SET StartTime = '14:00:00'
WHERE ClassID IN
   (SELECT ClassID
   FROM Faculty_Classes
   INNER JOIN Staff
   ON Faculty_Classes.StaffID = Staff.StaffID
   WHERE StfFirstName = 'Kathryn'
   AND StfLastName = 'Patterson')
```

So the trick is to identify the relationships between the target table and any related table(s) you need in order to specify the criteria in the WHERE clause.

You did this in Chapter 8, INNER JOINs, and Chapter 9, OUTER JOINs, as you assembled the FROM clause of queries on multiple tables. When building an UPDATE statement, you can put only the target table after the UPDATE keyword, so you must take the other tables and put them in a subquery that returns the column that you can link back to the target table.

Using a Subquery UPDATE Expression

If you thought we were done using subqueries, you were wrong. Notice in Figure 15–1 that the value that you can assign to a column in a SET clause can be a value expression. Just for review, Figure 15–7 shows how to construct a value expression.

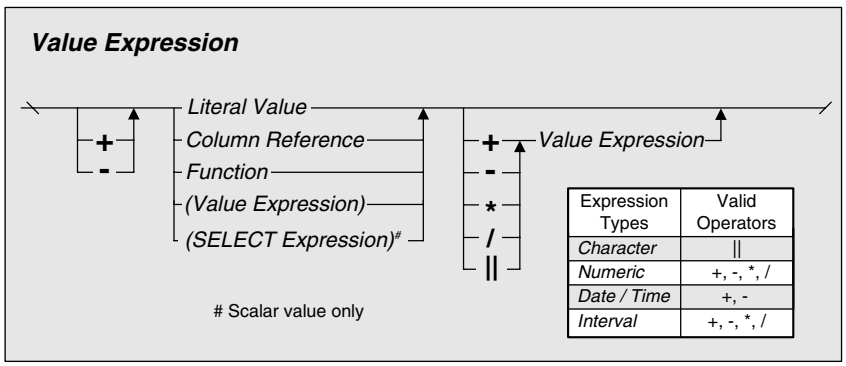


Figure 15–7 *The syntax diagram for a value expression*

In Chapter 2, Ensuring Your Database Structure Is Sound, we advised you to not include calculated fields in your tables. As with most rules, there are exceptions. Consider the Orders table in the Sales Orders sample database. If your business handles extremely large orders (thousands of order detail rows), you might want to consider including an order total field in the Orders table. Including this calculated field lets you run queries to examine the total of all items ordered without having to fetch and total thousands of detail rows. If you choose to do this, you must include code in your application that keeps the calculated total up to date every time a change is made to any related order detail row.

So far, we've been assigning a literal value or a value expression containing a literal value, an operator, and a column name to columns in the SET clause.

Notice that you can also assign the value of another column in the target table, but you'll rarely want to do that. The most interesting possibility is that you can use a SELECT expression (a subquery) that returns a single value (such as a sum) from another table and assign that value to your column. You can include criteria in the subquery (a WHERE clause) that filters the values from the other table based on a value in the table you're updating.

So, to update a total in one table (Orders) using a sum of an expression on columns in a related table (Order_Details), you can run an UPDATE query using a subquery. In the subquery, you'll sum the value of quantity ordered times quoted price and place it in the calculated field, and you'll add a WHERE clause to make sure you're summing values from related rows in the Order_Details table for each row in the Orders table. Your request might look like this.

"Change the orders table by setting the order total to the sum of quantity ordered times quoted price for all related order detail rows."

Translation: Update the orders table by setting the order total to the sum of quantity ordered times quoted price from the order details table where the order number matches the order number in the orders table

Clean Up:
Update ~~the~~ orders ~~table~~
~~by~~ set~~ting~~ ~~the~~ order total ~~to~~ =
~~the~~ (select sum ~~of~~ (quantity ordered ~~times~~ * quoted price)
from ~~the~~ order details ~~table~~
where ~~the~~ order_details.order number ~~matches the~~ =
orders.order number ~~in the orders table~~)

SQL:
```
UPDATE Orders
SET OrderTotal =
   (SELECT SUM(QuantityOrdered * QuotedPrice)
   FROM Order_Details
   WHERE Order_Details.OrderNumber =
   Orders.OrderNumber)
```

❖ **Note** We saved this query as CH15_Update_Order_Totals_Subquery in the Sales Orders Modify sample database.

Notice that we didn't include a WHERE clause to filter the orders that the database will update. If you execute this query in application code, you'll probably want to filter the order number so that the database updates only

the order that you know was changed. Some database systems actually let you define a calculated field like this and specify how the field should be updated by your database system. Most database systems also support something called a *trigger* that the database system runs on your behalf each time a row in a specified table is changed, added, or deleted. For systems that include these features, you can use this UPDATE query syntax in either the definition of the table or in the trigger you define to run when a value changes. As usual, consult your database documentation for details.

Uses for UPDATE

At this point, you should have a good understanding of how to update one or more columns in a table using either a simple literal or a complex subquery expression. You also know how to filter the rows that will be changed by your UPDATE statement. The best way to give you an idea of the wide range of uses for the UPDATE statement is to list some problems you can solve with this statement and then present a set of examples in the Sample Statements section.

"Reduce the quoted price by 2 percent for orders shipped more than 30 days after the order date."

"Add 6 percent to all agent salaries."

"Change the tournament location to 'Oasis Lanes' for all tournaments originally scheduled at 'Sports World Lanes.'"

"Recalculate the grade point average for all students based on classes completed."

"Apply a 5 percent discount to all orders for customers who purchased more than $50,000 in the month of October 2007."

"Correct the engagement contract price by multiplying the entertainer daily rate times number of days and adding a 15 percent commission."

"Update the city and state for all bowlers by looking up the names by ZIP Code."

"For all students and staff in zip codes 98270 and 98271, change the area code to 360."

"Make sure the retail price for all bikes is at least a 45 percent markup over the wholesale price of the vendor with the lowest cost."

"Apply a 2 percent discount to all engagements for customers who booked more than $3,000 worth of business in the month of October 2007."

"Change the name of the 'Huckleberrys' bowling team to 'Manta Rays.'"

"Increase the salary of full-time tenured staff by 5 percent."

"Set the retail price of accessories to the wholesale price of the highest priced vendor plus 35 percent."

"Add 0.5 percent to the commission rate of agents who have sold more than $20,000 in engagements."

"Calculate and update the total pins, games bowled, current average, and current handicap for all bowlers."

Sample Statements

You now know the mechanics of constructing UPDATE queries. Let's look at a set of samples, all of which request that one or more columns in a table be changed in some way. These examples come from four of the sample databases.

We've also included a view of each target table before and after executing the update and a count of the number of rows that should be changed by each sample UPDATE statement. The name that appears immediately before the count of rows changed is the name we gave each query in the sample data on the companion CD you'll find bound into the back of the book. We stored each query in the appropriate sample database (as indicated within the example) and prefixed the names of the queries relevant to this chapter with "CH15." You can follow the instructions in the Introduction of this book to load the samples onto your computer and try them.

❖ **Note** Remember that all the column names and table names used in these examples are drawn from the sample database structures shown in Appendix B, Schema for the Sample Databases. To simplify the process, we have combined the Translation and Clean Up steps for all the examples. These samples assume you have thoroughly studied and understood the concepts covered in previous chapters, especially the chapter on subqueries.

Sales Orders Database

"Reduce the quoted price by 2 percent for orders shipped more than 30 days after the order date."

Let's restate the problem so that it more closely follows the SQL syntax.

"Change order details by setting the quoted price to quoted price times 0.98 for all orders where the shipped date is more than 30 days later than the order date."

Translation/ Clean Up

Update ~~the~~ order details ~~table by~~
set~~ting the~~ quoted price ~~equal to~~ = ~~the~~ quoted price ~~times~~ * 0.98
where ~~the~~ order ID ~~is~~ in
~~the~~ (select~~ion of~~ order ID~~s~~
from ~~the~~ orders ~~table~~
where ship date ~~minus~~ - order date ~~is greater than~~ > 30

SQL

```
UPDATE Order_Details
SET QuotedPrice = QuotedPrice * 0.98
WHERE OrderID IN
   (SELECT OrderID
   FROM Orders
   WHERE (ShipDate - OrderDate) > 30)
```

❖ **Note** This query solution assumes your database system allows you to subtract one date from another to obtain the number of days between the two dates. Consult your database documentation for details.

Order_Details Table Before Executing the UPDATE Query

OrderNumber	ProductNumber	QuotedPrice	QuantityOrdered
291	1	$1,200.00	4
291	14	$139.95	2
291	30	$43.65	6
371	9	$32.01	6
371	22	$79.54	5
371	35	$37.83	6
387	1	$1,200.00	4
387	6	$635.00	4
<< more rows here >>			

Order_Details Table After Executing CH15_Adjust_Late_Order_Prices (29 rows changed)

OrderNumber	ProductNumber	QuotedPrice	QuantityOrdered
291	1	$1,176.00	4
291	14	$137.15	2
291	30	$42.78	6
371	9	$31.37	6
371	22	$77.95	5
371	35	$37.07	6
387	1	$1,176.00	4
387	6	$622.30	4
<< more rows here >>			

"Make sure the retail price for all bikes is at least a 45 percent markup over the wholesale price of the vendor with the lowest cost."

Restated, the request is as follows.

"Change the products table by setting the retail price equal to 1.45 times the wholesale price of the vendor that has the lowest cost for the product where the retail price is not already equal to 1.45 times the wholesale price and the category ID is 2."

Translation/ Clean Up

Update ~~the~~ products ~~table by~~
set~~ting the~~ retail price ~~equal to~~ = 1.45 ~~times~~ *
~~the~~ (select~~ion of the unique~~ distinct wholesale price
from ~~the~~ product vendors ~~table~~
where ~~the~~ product vendors ~~table's~~ product number
~~is equal to~~ = ~~the~~ products ~~table's~~ product number
and ~~the~~ wholesale price ~~is equal to~~ =
~~the~~ (select~~ion of the~~ min~~imum~~ (wholesale price)
from ~~the~~ product vendors ~~table~~
where ~~the~~ product vendors ~~table's~~ product number
~~is equal to~~ = ~~the~~ products ~~table's~~ product number))
where ~~the~~ retail price ~~is less than~~ < 1.45 times
~~the~~ (select~~ion of the unique~~ distinct wholesale price
from ~~the~~ product vendors ~~table~~
where ~~the~~ product vendors ~~table's~~ product number
~~is equal to~~ = ~~the~~ products ~~table's~~ product number
and ~~the~~ wholesale price ~~is equal to~~ =
~~the~~ (select~~ion of the~~ min~~imum~~ (wholesale price)
from ~~the~~ product vendors ~~table~~
where ~~the~~ product vendors ~~table's~~ product number
~~is equal to~~ = ~~the~~ products ~~table's~~ product number))
and ~~the~~ category ID ~~is equal to~~ = 2

SQL

```
UPDATE Products
SET RetailPrice = ROUND(1.45 *
   (SELECT DISTINCT WholeSalePrice
    FROM Product_Vendors
    WHERE Product_Vendors.ProductNumber
    = Products.ProductNumber
    AND WholeSalePrice =
      (SELECT MIN(WholeSalePrice)
       FROM Product_Vendors
       WHERE Product_Vendors.ProductNumber
       = Products.ProductNumber)), 0)
       WHERE RetailPrice < 1.45 *
          (SELECT DISTINCT WholeSalePrice
           FROM Product_Vendors
           WHERE Product_Vendors.ProductNumber
```

```
        = Products.ProductNumber
        AND WholeSalePrice =
            (SELECT MIN(WholeSalePrice)
             FROM Product_Vendors
             WHERE Product_Vendors.ProductNumber
             = Products.ProductNumber))
AND CategoryID = 2
```

❖ **Note** You'll find this query solved with a JOIN in the UPDATE clause in the Microsoft Access sample database because Access does not support a subquery in the SET clause.

Notice also that the solution rounds the resulting price to the nearest dollar (zero decimal places). You'll find that most commercial implementations support a ROUND function even though this function is not explicitly defined in the SQL Standard.

We could have also included a subquery to find the category ID that is equal to (or IN) the category IDs from the Categories table where category description is equal to Bikes, but we thought the query was complex enough without adding another subquery. Finally, we selected the DISTINCT wholesale price because more than one vendor might have the same low price. We want only one value from the subquery for the comparison.

Products Table Before Executing the UPDATE Query

Product Number	Product Name	Retail Price	Quantity OnHand	Category ID
2	Eagle FS-3 Mountain Bike	$1,800.00	8	2

Products Table After Executing CH15_Adjust_Bike_Retail_Price (1 row changed)

Product Number	Product Name	Retail Price	Quantity OnHand	Category ID
2	Eagle FS-3 Mountain Bike	$1,840.00	8	2

❖ **Note** If you scan the Product_Vendors table for all the bikes (product IDs 1, 2, 6, and 11), you'll find that only product 2 has a current retail price that is less than 1.45 times the lowest wholesale price for that product from any vendor. The wholesale price for bike 2 from vendor ID 6 is $1,269, and 1.45 times this amount is $1,840.05, which the query rounded to the nearest dollar.

Entertainment Agency Database

"Add 6 percent to all agent salaries."

Restated, the request is as follows.

"Change the agents table by adding 6 percent to all salaries."

Translation/ Clean Up: Update ~~the~~ agents ~~table~~ ~~by~~ set~~ting~~ salary ~~equal to~~ = salary ~~times~~ * 1.06

SQL:
```
UPDATE Agents
SET Salary = ROUND(Salary * 1.06, 0)
```

❖ **Note** We've again used the common ROUND function found in most commercial implementations and have specified rounding to zero decimal places. Check your database system documentation for specific details about rounding in your implementation.

Agents Table Before Executing the UPDATE Query

AgentID	AgtFirstName	AgtLastName	Salary
1	William	Thompson	$35,000.00
2	Scott	Bishop	$27,000.00
3	Carol	Viescas	$30,000.00
4	Karen	Smith	$22,000.00
5	Marianne	Wier	$24,500.00
6	John	Kennedy	$33,000.00
7	Caleb	Viescas	$22,100.00
8	Maria	Patterson	$30,000.00
9	Daffy	Dumbwit	$50.00

Agents Table After Executing CH15_Give_Agents_ 6Percent_Raise (9 rows changed)

AgentID	AgtFirstName	AgtLastName	Salary
1	William	Thompson	$37,100.00
2	Scott	Bishop	$28,620.00
3	Carol	Viescas	$31,800.00
4	Karen	Smith	$23,320.00
5	Marianne	Wier	$25,970.00
6	John	Kennedy	$34,980.00
7	Caleb	Viescas	$23,426.00
8	Maria	Patterson	$31,800.00
9	Daffy	Dumbwit	$53.00

"Correct the engagement contract price by multiplying the entertainer daily rate by the number of days and adding a 15 percent commission."

Let's restate that.

"Modify the engagements table by setting the contract price to 1.15 times the number of days for the contract times the entertainer daily rate."

Translation/ Clean Up

Update ~~the~~ engagements ~~table~~
~~by~~ set~~ting the~~ contract price ~~equal to~~ =
1.15 ~~times~~ * ~~the~~ (end date ~~minus~~ - ~~the~~ start date ~~plus~~ + 1)
~~and then times the~~ *
(select~~ion of the~~ entertainer price per day
from ~~the~~ entertainers ~~table~~
where ~~the~~ entertainers ~~table~~ entertainer ID ~~is equal to~~ =
~~the~~ engagements ~~table~~ entertainer ID

SQL

```
UPDATE Engagements
SET Engagements.ContractPrice =
ROUND(1.15 * (EndDate - StartDate + 1) *
   (Select EntPricePerDay
    FROM Entertainers
    WHERE Entertainers.EntertainerID =
    Engagements.EntertainerID), 0)
```

❖ **Note** This query solution assumes your database system allows you to subtract one date from another to obtain the number of days between the two dates. Consult your database documentation for details.

We add 1 to the difference to obtain the actual number of days because the entertainment occurs on both the first and the last days of the engagement. It's clear you need to do this for an engagement that is booked for only one day. The start and end days are the same, so the difference is zero, but the engagement played for exactly one day.

Entertainer Prices per Day

EntertainerID	EntStageName	EntPricePerDay
1001	Carol Peacock Trio	$175.00
1002	Topazz	$120.00
1003	JV & the Deep Six	$275.00
1004	Jim Glynn	$60.00
1005	Jazz Persuasion	$125.00
1006	Modern Dance	$250.00
1007	Coldwater Cattle Company	$275.00
1008	Country Feeling	$280.00
1009	Katherine Ehrlich	$145.00
1010	Saturday Revue	$250.00
1011	Julia Schnebly	$90.00
1012	Susan McLain	$75.00
1013	Caroline Coie Cuartet	$250.00

Engagements Table Before Executing the UPDATE Query

Engagement Number	Start Date	End Date	Contract Price	Customer ID	Agent ID	Entertainer ID
2	2007-09-01	2007-09-05	$200.00	10006	4	1004
3	2007-09-10	2007-09-15	$590.00	10001	3	1005
4	2007-09-11	2007-09-17	$470.00	10007	3	1004
5	2007-09-11	2007-09-14	$1,130.00	10006	5	1003
6	2007-09-10	2007-09-14	$2,300.00	10014	7	1008
7	2007-09-11	2007-09-18	$770.00	10004	4	1002
8	2007-09-18	2007-09-25	$1,850.00	10006	3	1007
9	2007-09-18	2007-09-28	$1,370.00	10010	2	1010
<< more rows here >>						

Engagements Table After Executing CH15_Calculate_Entertainment_ ContractPrice (111 rows changed)

Engagement Number	Start Date	End Date	Contract Price	Customer ID	Agent ID	Entertainer ID
2	2007-09-01	2007-09-05	$345.00	10006	4	1004
3	2007-09-10	2007-09-15	$862.00	10001	3	1005
4	2007-09-11	2007-09-17	$483.00	10007	3	1004
5	2007-09-11	2007-09-14	$1,265.00	10006	5	1003
6	2007-09-10	2007-09-14	$1,610.00	10014	7	1008
7	2007-09-11	2007-09-18	$1,104.00	10004	4	1002
8	2007-09-18	2007-09-25	$2,530.00	10006	3	1007
9	2007-09-18	2007-09-28	$3,162.00	10010	2	1010
<< more rows here >>						

❖ **Note** The original contract price values in the Engagements table are simply random values within a reasonable range that we chose when we created the original sample data. This update query clearly corrects each value to a more reasonable charge based on each entertainer's daily rate.

School Scheduling Database

"For all students in ZIP Codes 98270 and 98271, change the area code to 360.

Restated, the problem is as follows.

"Change the students table by setting the area code to 360 for all students who live in ZIP Codes 98270 and 98271."

Translation/ Clean Up
Update ~~the~~ students ~~table~~
~~by~~ set~~ting the~~ area code ~~equal to~~ = '360'
where ~~the~~ student ZIP Code ~~is~~ in ~~the list~~ ('98270', ~~and~~ '98271')

SQL

```
UPDATE Students
SET Students.StudAreaCode = '360'
WHERE Students.StudZipCode IN ('98270', '98271')
```

Students Table Before Executing the UPDATE Query

Student ID	StudFirst Name	StudLast Name	Stud City	Stud State	Stud ZipCode	Stud AreaCode
<< more rows here >>						
1007	Elizabeth	Hallmark	Marysville	WA	98271	253
1008	Sara	Sheskey	Portland	OR	97208	503
1009	Karen	Smith	Eugene	OR	97401	541
1010	Marianne	Wier	Tacoma	WA	98413	253
1011	John	Kennedy	Portland	OR	97208	503
1012	Sarah	Thompson	Lubbock	TX	79402	806
1013	Michael	Viescas	Redmond	WA	98052	425
1014	Kendra	Bonnicksen	Seattle	WA	98105	206
1015	Brannon	Jones	Long Beach	CA	90809	562
1016	Steve	Pundt	Dallas	TX	75204	972
1017	George	Chavez	Marysville	WA	98270	206
<< more rows here >>						

Students Table After Executing CH15_Fix_Student_AreaCode (2 rows changed)

Student ID	StudFirst Name	StudLast Name	StudCity	Stud State	StudZip Code	StudArea Code
<< more rows here >>						
1007	Elizabeth	Hallmark	Marysville	WA	98271	360
1008	Sara	Sheskey	Portland	OR	97208	503
1009	Karen	Smith	Eugene	OR	97401	541
1010	Marianne	Wier	Tacoma	WA	98413	253
1011	John	Kennedy	Portland	OR	97208	503
1012	Sarah	Thompson	Lubbock	TX	79402	806
1013	Michael	Viescas	Redmond	WA	98052	425
1014	Kendra	Bonnicksen	Seattle	WA	98105	206
1015	Brannon	Jones	Long Beach	CA	90809	562
1016	Steve	Pundt	Dallas	TX	75204	972
1017	George	Chavez	Marysville	WA	98270	360
<< more rows here >>						

"Recalculate the grade point average for all students based on classes completed."

Restated, the request looks like this.

"Modify the students table by setting the grade point average to the sum of the credits times the grade divided by the sum of the credits."

Translation/ Clean Up

Update ~~the~~ students ~~table~~
~~by~~ set~~ting the~~ student GPA ~~equal to~~ =
~~the~~ (select~~ion of the~~ sum ~~of~~ (credits ~~times~~ * grade)
~~divided by~~ / ~~the~~ sum ~~of~~ (credits)
from ~~the~~ classes ~~table~~
inner join~~ed with the~~ student schedules ~~table~~
on classes.class ID ~~in the classes table matches~~
= student_schedules.class ID ~~in the student schedules table~~
where ~~the~~ class status ~~is~~ = ~~complete~~ 2
and ~~the~~ student schedules ~~table~~ student ID ~~is equal to~~ =
~~the~~ students ~~table~~ student ID)

SQL

```
UPDATE Students
SET Students.StudGPA =
   (SELECT ROUND(SUM(Classes.Credits *
       Student_Schedules.Grade) /
       SUM(Classes.Credits), 3)
    FROM Classes
    INNER JOIN Student_Schedules
    ON Classes.ClassID = Student_Schedules.ClassID
    WHERE (Student_Schedules.ClassStatus = 2)
    AND (Student_Schedules.StudentID =
       Students.StudentID))
```

Students Table Before Executing the UPDATE Query

StudentID	StudFirstName	StudLastName	StudGPA
1001	Kerry	Patterson	74.465
1002	David	Hamilton	78.755
1003	Betsy	Stadick	85.235
1004	Janice	Galvin	81
1005	Doris	Hartwig	72.225
1006	Scott	Bishop	88.5
1007	Elizabeth	Hallmark	87.65
1008	Sara	Sheskey	84.625
<< more rows here >>			

Students Table After Executing the CH15_Update_Student_GPA Query (18 rows changed)

StudentID	StudFirstName	StudLastName	StudGPA
1001	Kerry	Patterson	73.875
1002	David	Hamilton	79.346
1003	Betsy	Stadick	87.737
1004	Janice	Galvin	80.78
1005	Doris	Hartwig	73.222
1006	Scott	Bishop	87.603
1007	Elizabeth	Hallmark	89
1008	Sara	Sheskey	85.726
<< more rows here >>			

❖ **Note** Because Microsoft Access does not support using subqueries with aggregate functions, you'll find this query solved as a series of calls to built-in functions using a predefined view on the Student_Schedules and Classes tables.

Bowling League Database

"Calculate and update the total pins, games bowled, current average, and current handicap for all bowlers."

> ❖ **Note** You calculated the handicap using a SELECT query in the Problems for You to Solve section of Chapter 13, Grouping Data. For a hint, see the CH13_Bowler_Average_Handicap query in the Bowling League sample database. Remember that the handicap is 90 percent of 200 minus the bowler's average.

Let's restate the problem like this.

"Modify the bowlers table by calculating the total pins, games bowled, current average, and current handicap from the bowler scores table."

Translation/ Clean Up

Update ~~the~~ bowlers ~~table~~
~~by~~ set~~ting the~~ total pins ~~equal to~~ =
~~the~~ (select~~ion of the~~ sum ~~of the~~ (raw score)
from ~~the~~ bowler scores ~~table~~
where ~~the~~ bowler scores ~~table~~ bowler ID
~~is equal to~~ = ~~the~~ bowlers ~~table~~ bowler ID),
~~and the~~ games bowled ~~equal to~~ =
~~the~~ (select~~ion of the~~ count ~~of the~~ (raw score)
from ~~the~~ bowler scores ~~table~~
where ~~the~~ bowler scores ~~table~~ bowler ID
~~is equal to~~ = ~~the~~ bowlers ~~table~~ bowler ID),
~~and the~~ current average ~~equal to~~ =
~~the~~ (select~~ion of the average~~ avg ~~of the~~ (raw score)
from ~~the~~ bowler scores ~~table~~
where ~~the~~ bowler scores ~~table~~ bowler ID
~~is equal to~~ = ~~the~~ bowlers ~~table~~ bowler ID),
~~and the~~ current handicap ~~equal to~~ =
~~the~~ (select~~ion of~~ 0.9 ~~times~~ * (200 ~~minus~~ –
~~the average~~ avg ~~of the~~ (raw score))
from ~~the~~ bowler scores ~~table~~
where ~~the~~ bowler scores ~~table~~ bowler ID
~~is equal to~~ = ~~the~~ bowlers ~~table~~ bowler ID)

```
SQL         UPDATE Bowlers
            SET Bowlers.BowlerTotalPins =
               (SELECT SUM(RawScore)
                FROM Bowler_Scores
                WHERE Bowler_Scores.BowlerID = Bowlers.BowlerID),
               Bowlers.BowlerGamesBowled =
               (SELECT COUNT(Bowler_Scores.RawScore)
                FROM Bowler_Scores
                WHERE Bowler_Scores.BowlerID = Bowlers.BowlerID),
                Bowlers.BowlerCurrentAverage =
                   (SELECT ROUND(AVG(Bowler_Scores.RawScore), 0)
                    FROM Bowler_Scores
                    WHERE Bowler_Scores.BowlerID =
                      Bowlers.BowlerID),
                Bowlers.BowlerCurrentHcp =
               (SELECT ROUND(0.9 *
                  (200 - ROUND(AVG(Bowler_Scores.RawScore),
                  0)), 0)
                FROM Bowler_Scores
                WHERE Bowler_Scores.BowlerID = Bowlers.BowlerID)
```

Bowlers Table Before Executing the UPDATE Query

Bowler ID	Bowler Last	Name Bowler First	Name Bowler Total Pins	Bowler Games Bowled	Bowler Current Average	Bowler Current Hcp
1	Fournier	Barbara	5790	39	148	47
2	Fournier	David	6152	39	158	38
3	Kennedy	John	6435	39	165	32
4	Sheskey	Sara	5534	39	142	52
5	Patterson	Ann	5819	39	149	46
6	Patterson	Neil	6150	39	158	38
7	Viescas	David	6607	39	169	28
8	Viescas	Stephanie	5558	39	143	51
<< more rows here >>						

Bowlers Table After Executing CH15_Calc_Bowler_Pins_Avg_Hcp (32 rows changed)

Bowler ID	Bowler Last	Name Bowler First	Name Bowler Total Pins	Bowler Games Bowled	Bowler Current Average	Bowler Current Hcp
1	Fournier	Barbara	6242	42	149	46
2	Fournier	David	6581	42	157	39
3	Kennedy	John	6956	42	166	31
4	Sheskey	Sara	5963	42	142	52
5	Patterson	Ann	6269	42	149	46
6	Patterson	Neil	6654	42	158	38
7	Viescas	David	7042	42	168	29
8	Viescas	Stephanie	5983	42	142	52
<< more rows here >>						

"Change the tournament location to 'Oasis Lanes' for all tournaments originally scheduled at 'Sports World Lanes.'"

Restated, the problem is as follows.

"Modify the tournaments table by changing the tournament location to 'Oasis Lanes' for all tournaments originally scheduled at 'Sports World Lanes.'"

Translation/ Clean Up: Update ~~the~~ tournaments ~~table~~ ~~by~~ set~~ting the~~ tourney location ~~equal to~~ = 'Oasis Lanes' where ~~the original~~ tourney location ~~is equal to~~ = 'Sports World Lanes'

SQL:

```
UPDATE Tournaments
SET TourneyLocation = 'Oasis Lanes'
WHERE TourneyLocation = 'Sports World Lanes'
```

Tournaments Table Before Executing the UPDATE Query

TourneyID	TourneyDate	TourneyLocation
5	2007-10-02	Sports World Lanes
12	2007-11-20	Sports World Lanes
18	2008-08-01	Sports World Lanes

Tournaments Table After Executing CH15_Change_Tourney_Location (3 rows changed)

TourneyID	TourneyDate	TourneyLocation
5	2007-10-02	Oasis Lanes
12	2007-11-20	Oasis Lanes
18	2008-08-01	Oasis Lanes

SUMMARY

We started the chapter with a brief discussion about the UPDATE statement used to change data in tables rather than to fetch data. We introduced the syntax of the UPDATE statement and explained a simple example to update one column in all the rows in a table using an expression.

Next we showed an example of how to use the WHERE clause to filter the rows you are updating. We also showed you how to construct a SELECT query first to verify that you'll be updating the correct rows, and we showed you how to map the clauses in your SELECT query into the UPDATE statement you need. Next we explained the importance of transactions and how you can use them to protect against errors or to ensure that either all changes or no changes are made to your tables. We continued our discussion by showing you how to update multiple columns in a table with a single UPDATE query.

Then we entered the realm of using subqueries in your UPDATE queries. We explained how to use a subquery to create a more complex filter in your WHERE clause. Finally, we showed you how to use a subquery to generate a

new value to assign to a column in your SET clause. The rest of the chapter provided examples of how to build UPDATE queries.

The following section presents a number of problems that you can work out on your own.

Problems for You to Solve

Below, we show you the request statement and the name of the solution query in the sample databases. If you want some practice, you can work out the SQL you need for each request and then check your answer with the query we saved in the samples. Don't worry if your syntax doesn't exactly match the syntax of the queries we saved—as long as your result is the same.

Sales Orders Database

1. *"Apply a 5 percent discount to all orders for customers who purchased more than $50,000 in the month of October 2007."*
   (Hint: You need a subquery within a subquery to fetch the order numbers for all orders where the customer ID of the order is in the set of customers who ordered more than $50,000 in the month of October.)
   You can find the solution in CH15_Give_Discount_To_Good_October_Customers (650 rows changed). Be sure to run CH15_Update_Order_Totals_Subquery to correct the totals in the Orders table after executing this query.
2. *"Set the retail price of accessories (category = 1) to the wholesale price of the highest-priced vendor plus 35 percent."*
   (Hint: See CH15_Adjust_Bike_Retail_Price in the Sample Statements for the technique.)
   You can find the solution in CH15_Adjust_Accessory_Retail_Price (11 rows changed).

Entertainment Agency Database

1. *"Apply a 2 percent discount to all engagements for customers who booked more than $3,000 worth of business in the month of October 2007."*
   (Hint: Use an aggregate subquery to find those customers with engagements in October HAVING total bookings greater than $3,000.)
   You can find the solution in CH15_Discount_Good_Customers_October (34 rows changed).
2. *"Add 0.5 percent to the commission rate of agents who have sold more than $20,000 in engagements."*

(Hint: Use an aggregate subquery to find those agents HAVING total bookings greater than $20,000.)
You can find the solution in CH15_Reward_Good_Agents (3 rows changed).

School Scheduling Database

1. *"Increase the salary of full-time tenured staff by 5 percent."*
   (Hint: Use a subquery in the WHERE clause to find matching staff IDs in the faculty table that have a status of full time and a tenured field value of true, that is, 1 or –1, depending on your database system.)
   You can find the solution in CH15_Give_FullTime_Tenured_Raise (21 rows changed.)
2. *"For all staff in ZIP Codes 98270 and 98271, change the area code to 360."*
   You can find the solution in CH15_Fix_Staff_AreaCode (2 rows changed).

Bowling League Database

1. *"Change the name of the 'Huckleberrys' bowling team to 'Manta Rays.'"*
   You can find the solution in CH15_Change_Huckleberry_Name (1 row changed).
2. *"Update the city and state for all bowlers by looking up the names by ZIP Code."*
   (Hint: Use a subquery to fetch the matching city name and another subquery to fetch the matching state from the WAZips table.)
   You can find the solution in CH15_Update_Bowler_City_State (6 rows changed).

This page intentionally left blank

16

Inserting Sets of Data

"I was brought up to believe that the only thing worth doing was to add to the sum of accurate information in the world."
—Margaret Meade

Topics Covered in This Chapter

What Is an INSERT?

The INSERT Statement

Uses for INSERT

Sample Statements

Summary

Problems for You to Solve

To this point, you have learned how to fetch information from your tables in creative ways. In the previous chapter, you learned how to modify existing data by using the UPDATE statement. But how do you put data into your tables to begin with? The data certainly doesn't appear magically on its own! You'll learn the answer in this chapter—how to use the INSERT statement to add rows into your tables.

What Is an INSERT?

Most commercial database systems come with one or more graphical interface programs that let you work with data displayed on your screen. For example, you can open any table in Microsoft Office Access by simply finding the table object and opening it. Access displays the data in something it calls a datasheet that looks like a grid with columns and rows. You scroll to the end of the display to find a blank row, type data into the columns on that row, and

then move to another row to insert a new row in your table. You can also use Access to work with tables in Microsoft SQL Server in the same manner. You can do something similar using the MySQL query browser, and IBM's DB2 and Oracle Corporation's Oracle database provide equivalent tools.

But what's really happening when you type in new data and tell the system to save it? The graphical interface tools actually execute a command using SQL to add the data you just entered to your table. The SQL statement that these programs use is INSERT. If you browse through the sample files, you can find scripts that we generated to load the data into the sample databases. For example, the first few lines of the EntertainmentAgencyData.SQL file look like this:

```
USE EntertainmentAgencyExample;
INSERT INTO Customers
  VALUES (10001, 'Doris', 'Hartwig', '4726 - 11th Ave. N.E.',
   'Seattle', 'WA', '98105', '555-2671');
INSERT INTO Customers
  VALUES (10002, 'Deb', 'Waldal', '908 W. Capital Way',
   'Tacoma', 'WA', '98413', '555-2496');
INSERT INTO Customers
  VALUES (10003, 'Peter', 'Brehm', '722 Moss Bay Blvd.',
   'Kirkland', 'WA', '98033', '555-2501');
INSERT INTO Customers
  VALUES (10004, 'Dean', 'McCrae', '4110 Old Redmond Rd.',
   'Redmond', 'WA', '98052', '555-2506');
INSERT INTO Customers
  VALUES (10005, 'Elizabeth', 'Hallmark', 'Route 2, Box 203B',
   'Auburn', 'WA', '98002', '555-2521');
```

The first command (USE) tells the database system which database to use for the following commands. Each INSERT statement tells the database system to add exactly one row to a specific table. This might seem like a tedious process to load thousands of records into a sample database, but you'll find that each script to load data actually runs in just a few seconds. For some of the simpler tables, we used the graphical user interface to directly type in the data. To generate data for other sample tables, we wrote some application code to create and execute the INSERT statements. If you're familiar with Microsoft Access and Visual Basic, you can find code to generate sample data in the zfrmSell-Products form in the Sales Orders sample database.

If you write any applications, whether for desktop systems or for the Web, you'll create code to generate and execute the appropriate INSERT statement

when your user enters new data. Most of the time, you'll use the INSERT . . . VALUES version to add the data. In this chapter, you'll also learn about a second form of the INSERT statement that makes it easy to copy data from one table to another.

❖ **Note** You can find all the sample statements and solutions in the "modify" version of the respective sample databases—SalesOrdersModify, EntertainmentAgencyModify, SchoolSchedulingModify, and BowlingLeagueModify.

The INSERT Statement

SQL has two main versions of the INSERT statement. In the first version, you include the VALUES keyword and list the values that you want your database system to add as a single new row in a specified target table. The second version lets you use a SELECT clause to fetch data from a table to insert into your target table. Let's take a look at the VALUES version first.

Inserting Values

Although SQL is primarily designed to work with sets of data, much of the time you'll use INSERT to add a single row of data to one of your tables. The simplest way to add one row to a table is to use the INSERT statement with the VALUES clause. Figure 16–1 shows the diagram for this statement.

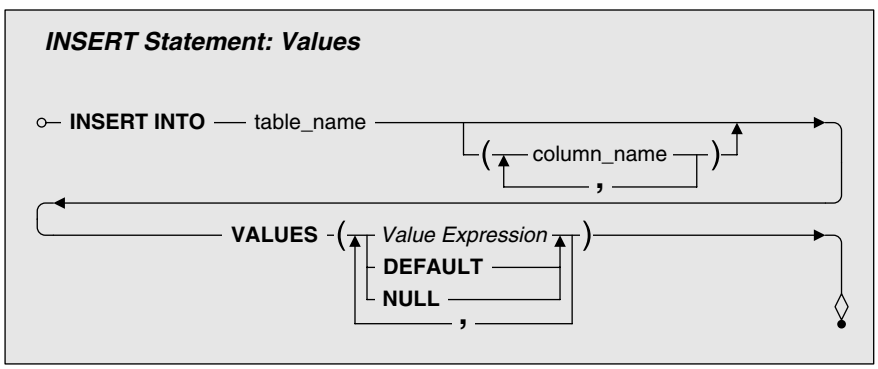


Figure 16–1 *The syntax diagram of the INSERT statement using the VALUES clause*

As you can see, you begin the statement with the INSERT INTO keywords. Next, specify the name of the table where you want to add the row. If you're going to supply values for all the columns, you can omit the column name list. (For example, we didn't specify the column name list in the INSERT statements we use to load the sample data because we're supplying a value for every column.) However, even when you plan to supply values for all columns, we recommend that you include the list of columns for which you intend to specify a data value. If you don't do that, your query will break if someone later adds a column to the table definition or changes the sequence of columns in the table. You specify the column name list by entering a left parenthesis, the column names separated by commas if you specify more than one, and a closing right parenthesis.

❖ **Note** The SQL Standard indicates that table_name can also be a view name, but the view must be "updatable and insertable." Many database systems support inserting rows into views, and each database system has its own rules about what constitutes an updatable or insertable view.

In most cases, a view isn't insertable if one of the output columns is the result of an expression or an aggregate function. Some database systems also support defining the view using JOIN and ON keywords in place of table_name. Consult your database system documentation for details. In this chapter, we'll exclusively use a single table as the target for each INSERT statement.

Finally, specify the VALUES keyword, a left parenthesis, a list of value expressions separated by commas, and a closing right parenthesis. Note that you must specify each value in the same sequence that you specified them in the column name list. That is, the first value expression supplies the value for the first column in the list, the second value expression for the second column in the list, and so on. If you're including values for all columns and did not include the column name list, your values must be in the same sequence as the columns in the table definition. If you want your database system to use the default value defined for a column, use the DEFAULT keyword. (But you'll get an error if no default value is defined.) To supply the Null value, use the NULL keyword.

Remember from earlier chapters that a value expression can be quite complex and can even include a subquery to fetch a single value from the target table or another table. For review, Figure 16–2 shows the diagram of a value expression.

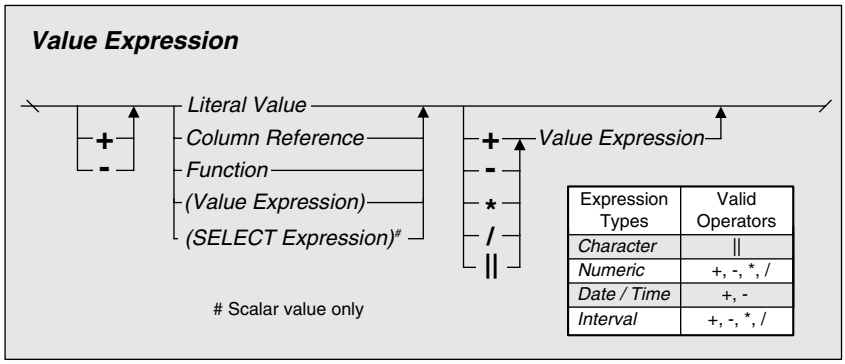


Figure 16–2 *Use a value expression, whose syntax is shown here, in a VALUES clause to specify the value of each column in your target table.*

❖ **Note** Not all database systems allow you to use a SELECT expression in the VALUES clause of an INSERT statement. Check your database documentation for details.

Let's look at how to add one row to the Employees table in the Sales Orders sample database. As with all queries, you should know the structure of the table. Figure 16–3 shows the design of the Employees table.

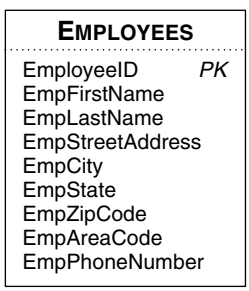


Figure 16–3 *The Employees table in the Sales Orders sample database*

Now let's formulate a request.

❖ **Note** Throughout this chapter, we use the "Request/Translation/Clean Up/SQL" technique introduced in Chapter 4, Creating a Simple Query.

"Add new employee Susan Metters at 16547 NE 132nd St, Woodinville, WA 98072, with area code 425 and phone number 555-7825."

You typically won't list the columns you need in your original request, but keep them in mind as you go to the Translation step. Here's how you might translate the request to add a new employee row.

Translation	Insert into the employees table in the columns first name, last name, street address, city, state, ZIP Code, area code, and phone number the values Susan, Metters, 16547 NE 132nd St, Woodinville, WA, 98072, 425, and 555-7825
Clean Up	Insert into ~~the~~ employees ~~table in the columns~~ (first name, last name, street address, city, state, ZIP Code, area code, ~~and~~ phone number) ~~the~~ values ('Susan', 'Metters', '16547 NE 132nd St', 'Woodinville', 'WA', '98072', 425, ~~and~~ '555-7825')

SQL

```
INSERT INTO Employees
   (EmpFirstName, EmpLastName,
   EmpStreetAddress, EmpCity, EmpState,
   EmpZipCode, EmpAreaCode, EmpPhoneNumber)
VALUES ('Susan', 'Metters',
   '16547 NE 132nd St', 'Woodinville', 'WA',
   '98072', 425, '555-7825')
```

You can find this query saved as CH16_Add_Employee in the modify version of the Sales Orders sample database.

Are you wondering why we didn't include the primary key (EmployeeID) of the Employees table? If so, read on!

Generating the Next Primary Key Value

In the example query in the previous section, we didn't include the primary key—EmployeeID. In all database systems, the primary key must have a value. Won't this query fail?

The answer is no, but only because we took advantage of a special feature that you'll find in nearly all commercial database implementations. When you're not concerned about the value of the primary key in a table—except that the value must be unique—you can usually define the primary key using a special

data type that the database system will increment for you every time you insert a new row. In Microsoft Access, use the data type called AutoNumber. (The data type is actually an integer with special attributes.) In Microsoft SQL Server, use the Identity data type (also an integer). For MySQL, use an integer with the special AUTO_INCREMENT attribute. The SQL syntax used in our example works in all three types of sample databases because we used this special feature for the primary key fields in nearly all the sample tables in the modify versions.

The Oracle database system is a bit different. Rather than provide a special data type, Oracle lets you define a Sequence pseudo-column, and you reference the NEXTVAL property of the pseudo-column every time you need a unique value for a new row. In Oracle, let's assume you previously defined a pseudo-column called EmpID. You can write your SQL like this:

```
SQL         INSERT INTO Employees
            (EmployeeID, EmpFirstName, EmpLastName,
            EmpStreetAddress, EmpCity, EmpState,
            EmpZipCode, EmpAreaCode, EmpPhoneNumber)
            VALUES (EmpID.NEXTVAL, 'Susan', 'Metters',
            '16547 NE 132nd St', 'Woodinville', 'WA',
            '98072', 425, '555-7825')
```

Note that we're now providing a value for each column in the table in the sequence that the columns are defined in the table definition. We could eliminate the optional column name list and write the SQL like this:

```
SQL         INSERT INTO Employees
            VALUES (EmpID.NEXTVAL, 'Susan', 'Metters',
            '16547 NE 132nd St', 'Woodinville', 'WA',
            '98072', 425, '555-7825')
```

As noted earlier, we don't recommend that you omit the column name list because your query will fail if your database administrator adds a column or changes the sequence of the column definitions. We present this option only for completeness.

If you really have your thinking cap on, you might wonder whether you could just as easily generate the next value by using a subquery expression. The SQL standard certainly supports this, and your SQL might look like this:

```
SQL         INSERT INTO Employees
            (EmployeeID,
            EmpFirstName, EmpLastName,
            EmpStreetAddress, EmpCity, EmpState,
            EmpZipCode, EmpAreaCode, EmpPhoneNumber)
            VALUES (
            (SELECT MAX(EmployeeID) FROM Employees) + 1,
            'Susan', 'Metters',
            '16547 NE 132nd St', 'Woodinville', 'WA',
            '98072', 425, '555-7825')
```

Unfortunately, several of the major database systems do not yet support a subquery in a VALUES clause. Check your database documentation for details.

Inserting Data by Using SELECT

Because we've focused so far on inserting only a single row at a time, you're probably wondering why we named this chapter "Inserting Sets of Data." In one sense, values for multiple columns in one row is a set of data, but you probably think of a set as consisting of multiple rows. Never fear—you can also insert a set of rows by using a SELECT expression in place of the VALUES clause. Because a SELECT expression fetches rows from one or more tables, you can think of an INSERT statement with SELECT as a powerful way to copy data. Figure 16–4 shows the syntax diagram for an INSERT statement using a SELECT expression.

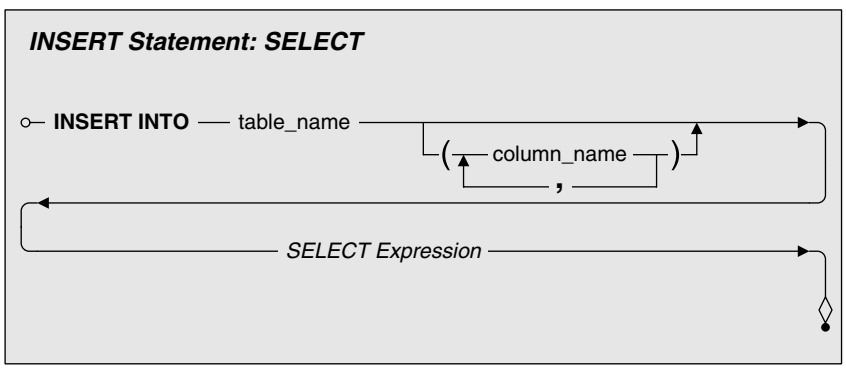


Figure 16–4 *The syntax diagram of the INSERT statement using a SELECT expression*

Notice that this variant of the INSERT statement begins in the same way. Following the INSERT INTO keywords, specify the name of the table that is the

target of this insert. If your SELECT expression returns the same number of columns and in the same order as in your target table, you can omit the optional column name list. But as we recommended earlier, even when you plan to supply values for all columns, we recommend that you include the list of columns for which you intend to specify a data value. If you don't do that, your query will break if someone later adds a column to the table definition or changes the sequence of columns in the table.

If you examine the SQL diagrams in Appendix A, SQL Standard Diagrams, you'll find that a SELECT expression is simply a SELECT statement that is optionally combined with additional SELECT statements using the UNION, INTERSECT, or EXCEPT operations. (See Chapter 7, Thinking in Sets, for an explanation of these three operations and Chapter 10, UNIONs, for a detailed description of UNION.) Figure 16–5 shows the syntax diagram for a SELECT statement.

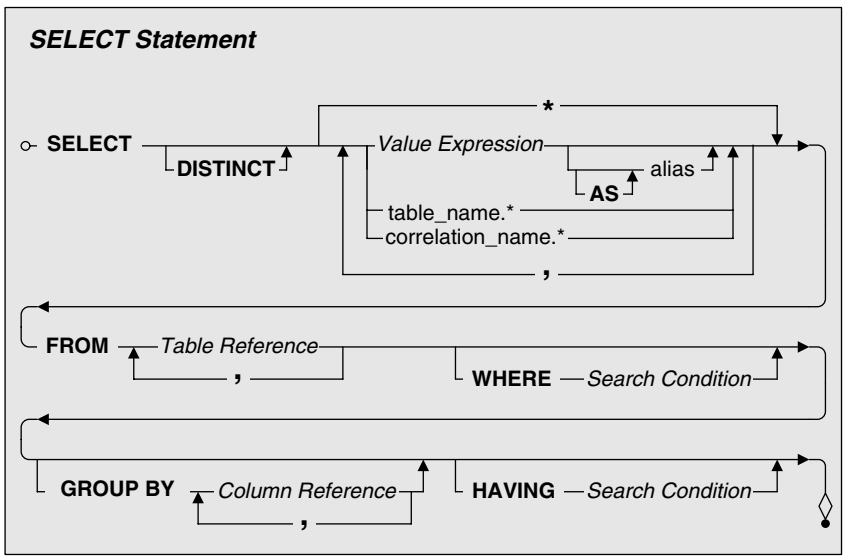


Figure 16–5 *The syntax diagram of a SELECT statement*

You might recall from earlier chapters that a table reference can be a single table name, a list of tables separated by commas, or a complex JOIN of two or more tables. A search condition can be a simple comparison of a column to a value; a more complex test using BETWEEN, IN, LIKE, or NULL; or a very complex predicate using subqueries. In short, you have all the power of SELECT

queries that you've learned about in earlier chapters at your disposal to specify the set of rows that you want to copy to a table.

Let's dig in and work through some examples that you can solve using INSERT with a SELECT expression. Here's a simple request that requires copying the data from a row in one table into another table.

> *"We just hired customer David Smith. Copy to the Employees table all the details for David Smith from the Customers table."*

As when building any query, you need to be familiar with the structure of the tables involved. Figure 16–6 shows the design of the two tables.

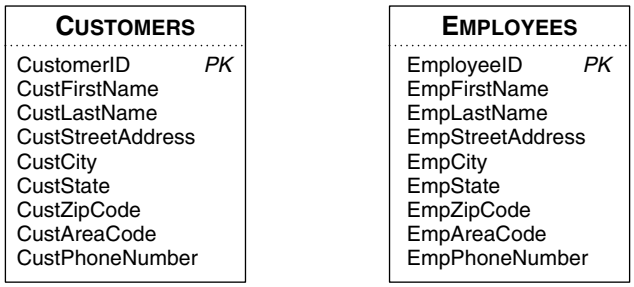


Figure 16–6 *The Customers and Employees tables in the Sales Orders sample database*

Let's restate the request so that it's easier to translate into an INSERT query.

> *"Copy to the Employees table the relevant columns in the Customers table for customer David Smith."*

Translation	Insert into the employees table in the columns first name, last name, street address, city, state, ZIP code, area code, and phone number the selection of the first name, last name, street address, city, state, ZIP code, area code, and phone number columns from the customers table where the customer first name is 'David' and the customer last name is 'Smith'
Clean Up	Insert into ~~the~~ employees ~~table~~ ~~in the columns~~ (first name, last name, street address, city, state, ZIP code, area code, ~~and~~ phone number) ~~the~~ select~~ion of the~~ first name, last name, street address, city, state, ZIP code,

```
                area code, and phone number columns
                from the customers table
                where the customer first name is = 'David'
                and the customer last name is = 'Smith'
SQL             INSERT INTO Employees
                   (EmpFirstName, EmpLastName, EmpStreetAddress,
                      EmpCity, EmpState, EmpZipCode,
                      EmpAreaCode, EmpPhoneNumber)
                   SELECT Customers.CustFirstName,
                      Customers.CustLastName,
                      Customers.CustStreetAddress,
                      Customers.CustCity,
                      Customers.CustState, Customers.CustZipCode,
                      Customers.CustAreaCode,
                      Customers.CustPhoneNumber
                   FROM Customers
                   WHERE (Customers.CustFirstName = 'David')
                   AND (Customers.CustLastName = 'Smith')
```

Notice that we did not include the EmployeeID column because we're depending on the database system to generate the next unique value for the new row(s) being inserted. You can find this query saved as CH16_Copy_Customer_To_Employee in the modify version of the Sales Orders sample database.

Because there's only one customer named David Smith, this query copies exactly one row to the Employees table. This still isn't a set of rows, but you can see how easy it is to use a SELECT expression to fetch the values you need to insert when they're available in another table.

Let's move on to a problem that could potentially insert hundreds of rows. In every active database application that collects new information over time, you might want to design a feature that allows the user to archive or copy to a backup table all transactions that occurred at some point in the past. The idea is that you don't want old historical data slowing down the processing of new data by making your application wade through thousands of rows that represent transactions that occurred long ago.

So, you might want to write an INSERT statement that copies transactions that happened earlier than a specific date into a table reserved for historical data. (In the next chapter, we'll show you how to delete the copied or archived transactions from the active table.) A typical request might look like this.

"Archive all engagements earlier than January 1, 2008."

In this particular case, both the Engagements table and the Engagements_Archive table have the same design, as shown in Figure 16–7.

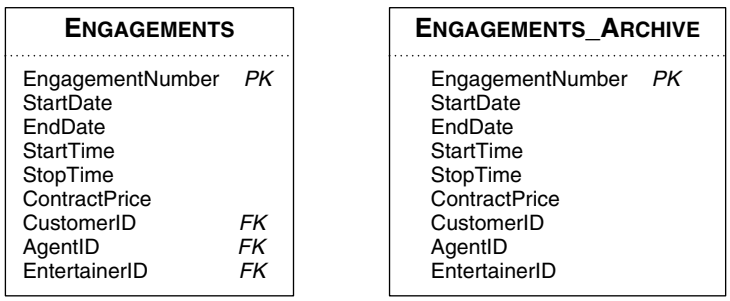


Figure 16–7 *The Engagements and Engagements_Archive tables in the Entertainment Agency sample database*

This is one case where you can safely leave out the column name list. The translation is very easy, and it looks like this.

Translation	Insert into the engagements archive table the selection of all columns from the engagements table where the engagement end date is earlier than January 1, 2008
Clean Up	Insert into ~~the~~ engagements archive ~~table~~ ~~the~~ select~~ion of all columns~~ * from ~~the~~ engagements ~~table~~ where ~~the~~ engagement end date ~~is earlier than~~ < ~~January 1, 2008~~ '2008-01-01'

SQL	`INSERT INTO Engagements_Archive`
	`  SELECT Engagements.*`
	`  FROM Engagements`
	`  WHERE Engagements.EndDate`
	`  < '2008-01-01'`

That's almost too easy, right? But remember that we recommended that you always explicitly list the column names. If you do that, your query will still run even if someone adds a new column to either table or changes the sequence of the columns. It's a bit more effort, but we recommend writing your SQL for this problem to look like this.

```
SQL         INSERT INTO Engagements_Archive
               (EngagementNumber, StartDate, EndDate,
                StartTime, StopTime, ContractPrice,
                CustomerID, AgentID, EntertainerID)
               SELECT Engagements.EngagementNumber,
                  Engagements.StartDate, Engagements.EndDate,
                  Engagements.StartTime, Engagements.StopTime,
                  Engagements.ContractPrice,
                  Engagements.CustomerID,
                  Engagements.AgentID, Engagements.EntertainerID
               FROM Engagements
               WHERE Engagements.EndDate < '2008-01-01'
```

You'll find this query saved as CH16_Archive_Engagements in the modify version of the Entertainment Agency sample database.

Now let's look at a creative way to use a SELECT expression. Consider the following request.

> *"Add a new product named 'Hot Dog Spinner' with a retail price of $895 in the Bikes category."*

You can see the tables you need in Figure 16–8.

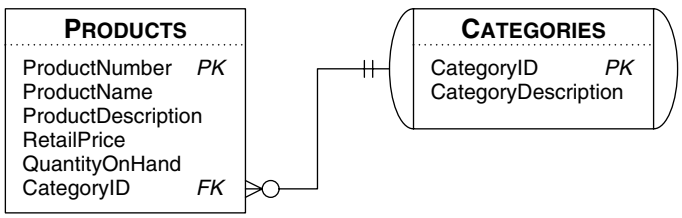


Figure 16–8 *The Products table and the related Categories table in the Sales Orders database*

Your target table is clearly the Products table, but that table requires a numeric value in the CategoryID field. The request says "in the Bikes category," so how do you suppose you can find the related CategoryID that you need for the Products table? Use a SELECT expression! You want to supply values also for the ProductName and RetailPrice columns, but remember that a SELECT statement can include literal values for some or all output columns. So you can fetch the related category ID from the Categories table and supply the other values you intend to insert as literal values. Let's restate the request and then solve it.

"Add a row to the products table using the values 'Hot Dog Spinner' for the product name, $895 for the retail price, and the category ID from the categories table for the category 'Bikes.'"

Translation
Insert into the products table in the columns product name, retail price, and category ID the selection of 'Hot Dog Spinner' as the product name, 895 as the retail price, and category ID from the categories table where the category description is equal to 'Bikes'

Clean Up
Insert into ~~the~~ products ~~table~~
~~in the columns~~ (product name, retail price, ~~and~~ category ID)
~~the~~ select~~ion of~~ 'Hot Dog Spinner' as ~~the~~ product name,
895 as ~~the~~ retail price, ~~and~~ category ID
from ~~the~~ categories ~~table~~
where ~~the~~ category description ~~is equal to~~ = 'Bikes'

SQL

```
INSERT INTO Products
(ProductName, RetailPrice, CategoryID)
SELECT 'Hot Dog Spinner' AS ProductName,
895 AS RetailPrice, CategoryID
FROM Categories
WHERE CategoryDescription = 'Bikes'
```

You might think using a SELECT Expression is useful only for copying entire rows, but as you have just seen, it's also useful to fetch one or more discrete values from a table that can supply the values you need. You'll find some interesting applications of this technique in the Sample Statements section later in this chapter.

Uses for INSERT

At this point, you should have a good understanding of how to insert one or more rows in a table using either a simple VALUES clause or a SELECT expression. The best way to give you an idea of the wide range of uses for the INSERT statement is to list some problems you can solve with this statement and then present a set of examples in the Sample Statements section. Here's just a small list of the types of problems you can solve with INSERT.

"Create a new bowler record for Matthew Patterson by copying the record for Neil Patterson."

"In the Entertainment database, create a new customer record for Kendra Hernandez at 457 211th St NE, Bothell, WA 98200, with a phone number of 555-3945."

"In the Sales Order database, create a new customer record for Mary Baker at 7834 W 32nd Ct, Bothell, WA 98011, with area code 425 and phone number 555-9876."

"Create a new subject category called 'Italian' with a subject code of 'ITA' in the Humanities department."

"Add a new team called the 'Aardvarks' with no captain assigned."

"Add a new engagement for customer Matt Berg booking entertainer Jazz Persuasion from 7 PM to 11 PM on August 15 and 16, 2008, that was booked by agent Karen Smith."

"Add a new vendor named Hot Dog Bikes at 1234 Main Street, Chicago, IL 60620, with phone number (773) 555-6543, fax number (773) 555-6542, website address http://www.hotdogbikes.com/, and e-mail address Sales@hotdogbikes.com."

"Add a new class for subject ID 4 (Intermediate Accounting) to be taught in classroom 3315 for 5 credits starting at 3:00 PM for 80 minutes on Tuesdays and Thursdays."

"Archive the tournament, tourney match, match game, and bowler scores for all matches played in 2007."

"Add 'New Age' to the list of musical styles."

"Archive all orders and order details for orders placed before January 1, 2008."

"Angel Kennedy wants to register as a student. Her husband is already enrolled. Create a new student record for Angel using the information from John's record."

"Duplicate all the tournaments and tourney matches played in 2007 for the same week in 2009."

"Agent Marianne Wier would like to book some entertainers, so create a new customer record by copying relevant fields from the agents table."

"Customer Liz Keyser wants to order again the products ordered on December 11, 2007. Use June 12, 2008, as the order date and June 15, 2008, as the shipped date."

"Staff member Tim Smith wants to enroll as a student. Create a new student record from Tim's staff record."

"Customer Doris Hartwig would like to rebook the entertainers she hired to play on December 1, 2007, for August 4, 2008."

"Customer Angel Kennedy wants to order again all the products ordered during the month of November 2007. Use June 15, 2008 as the order date and June 18, 2008, as the shipped date."

Sample Statements

You now know the mechanics of constructing INSERT queries. Let's look at a set of samples, all of which request that one or more rows be added to a table. These examples come from four of the sample databases.

We've also included a view of the data that the sample INSERT statement should add to the target table and a count of the number of rows that should be added. The name that appears immediately before the count of rows inserted is the name we gave each query in the sample data on the companion CD you'll find bound into the back of the book. We stored each query in the appropriate sample database (as indicated within the example) and prefixed the names of the queries relevant to this chapter with "CH16." You can follow the instructions in the Introduction of this book to load the samples onto your computer and try them.

❖ **Note** Remember that all the column names and table names used in these examples are drawn from the sample database structures shown in Appendix B, Schema for the Sample Databases. To simplify the process, we have combined the Translation and Clean Up steps for all the examples. These samples assume you have thoroughly studied and understood the concepts covered in previous chapters.

Sales Orders Database

"Add a new vendor named Hot Dog Bikes at 1234 Main Street, Chicago, IL 60620, with phone number (773) 555-6543, fax number (773) 555-6542, Web site address http://www.hotdogbikes.com/, and e-mail address Sales@hotdogbikes.com."

Translation/ Clean Up

Insert into ~~the~~ vendors ~~table~~
~~in the columns~~ (VendName, VendStreetAddress, VendCity, VendState, VendZipCode, VendPhoneNumber, VendFaxNumber, VendWebPage, ~~and~~ VendEMailAddress)
~~the~~ values ('Hot Dog Bikes', '1234 Main Street', 'Chicago', 'IL', '60620', '(773) 555-6543', '(773) 555-6542', 'http://www.hotdogbikes.com/', ~~and~~ 'Sales@hotdogbikes.com')

SQL

```
INSERT INTO Vendors
   (VendName, VendStreetAddress, VendCity,
    VendState, VendZipCode, VendPhoneNumber,
    VendFaxNumber, VendWebPage,
    VendEMailAddress)
VALUES ('Hot Dog Bikes', '1234 Main Street',
'Chicago',
    'IL', '60620', '(773) 555-6543',
    '(773) 555-6542', 'http://www.hotdogbikes.com/',
    'Sales@hotdogbikes.com')
```

Row Inserted into the Vendors Table by the CH16_Add_Vendor Query (1 row added)

Vend Name	Vend Street Address	Vend City	Vend State	Vend ZipCode	Vend Phone Number	Vend Fax Number	Vend Web Page	Vend EMail Address
Hot Dog Bikes	1234 Main Street	Chicago	IL	60620	(773) 555-6543	(773) 555-6542	http://www.hotdogbikes.com/	Sales@hotdogbikes.com

"Archive all orders and order details for orders placed before January 1, 2008."

❖ **Note** To archive all the information about an order, you need to copy data from both the Orders and the Order_Details tables, so you need two queries. Be sure to run the INSERT query for the orders first because rows in the Orders_Details_Archive table have foreign keys in the OrderID column that points to the same column in the Orders_Archive table.

If your system supports transactions (see the discussion in Chapter 15, Updating Sets of Data), you can start a transaction, run the query to copy orders followed by the query to copy order details, and then commit both INSERT actions if both ran with no errors. If the second query causes an error, you can roll back the transaction, which will ensure that none of the orders rows are copied. There's no point in copying only half the information about orders.

Because you're archiving rows by date, the query to archive order details must use a subquery filter for all order ID values that appear in the Orders table before the specified date.

Translation 1 / Clean Up	Insert into ~~the~~ orders archive ~~table~~ ~~the~~ select~~ion of~~ order number, order date, ship date, customer ID, employee ID, ~~and~~ order total from ~~the~~ orders ~~table~~ where ~~the~~ order date ~~is earlier than~~ < '2008-01-01'

SQL	`INSERT INTO Orders_Archive`
	`  SELECT OrderNumber, OrderDate, ShipDate,`
	`  CustomerID, EmployeeID, OrderTotal`
	`  FROM Orders`
	`  WHERE OrderDate < '2008-01-01'`

Translation 2 / Clean Up

Insert into ~~the~~ order details archive ~~table~~
~~the~~ select~~ion of~~ order number, product number,
quoted price, ~~and~~ quantity ordered
from ~~the~~ order details ~~table~~
where ~~the~~ order number ~~is~~ in
~~the~~ select~~ion of the~~ order number
from ~~the~~ orders ~~table~~
where ~~the~~ order date ~~is earlier than~~ < '2008-01-01')

SQL

```
INSERT INTO Order_Details_Archive
   SELECT OrderNumber, ProductNumber,
      QuotedPrice, QuantityOrdered
   FROM Order_Details
   WHERE Order_Details.OrderNumber IN
      (SELECT OrderNumber
       FROM Orders
       WHERE Orders.OrderDate < '2008-01-01')
```

Rows Inserted into the Orders_Archive Table by the CH16_Archive_2007_ Orders Query (598 rows added)

OrderNumber	OrderDate	ShipDate	CustomerID	EmployeeID	OrderTotal
1	2007-09-01	2007-09-04	1018	707	$12,751.85
2	2007-09-01	2007-09-03	1001	703	$816.00
3	2007-09-01	2007-09-04	1002	707	$11,912.45
4	2007-09-01	2007-09-03	1009	703	$6,601.73
5	2007-09-01	2007-09-01	1024	708	$5,544.75
6	2007-09-01	2007-09-05	1014	702	$9,820.29
7	2007-09-01	2007-09-04	1001	708	$467.85
8	2007-09-01	2007-09-01	1003	703	$1,492.60
<< more rows here >>					

❖ **Note** Neither query follows our recommendation to always include the column name list, but we wrote these two queries this way to show you examples where the column name list is not absolutely required.

Entertainment Agency Database

"Create a new customer record for Kendra Hernandez at 457 211th St NE, Bothell, WA 98200, with a phone number of 555-3945."

Translation/ Clean Up

Insert into ~~the~~ customers ~~table~~
~~in the columns~~ (customer first name, customer last name, customer street address, customer city, customer state, customer ZIP Code, ~~and~~ customer phone number)
~~the~~ values ('Kendra', 'Hernandez', '457 211th St NE', 'Bothell', 'WA', '98200', ~~and~~ '555-3945')

SQL

```
INSERT INTO Customers
   (CustFirstName, CustLastName,
    CustStreetAddress, CustCity, CustState,
    CustZipCode, CustPhoneNumber)
VALUES ('Kendra', 'Hernandez',
   '457 211th St NE', 'Bothell', 'WA',
   '98200', '555-3945')
```

Row Inserted into the Customers Table by the CH16_Add_Customer Query (1 row added)

CustFirst Name	CustLast Name	CustStreet Address	Cust City	Cust State	Cust ZipCode	CustPhone Number
Kendra	Hernandez	457 211th St NE	Bothell	WA	98200	555-3945

"Add a new engagement for customer Matt Berg booking entertainer Jazz Persuasion from 7 P.M. to 11 P.M. on August 15 and 16, 2008, which was booked by agent Karen Smith."

❖ **Note** If you look at the Engagements table, you'll find that you need the customer ID for Matt Berg from the Customers table, the entertainer ID for Jazz Persuasion from the Entertainers table, and the agent ID for Karen Smith from the Agents table. You can fetch these values by using a SELECT expression.

Be careful that you include the three tables you need in the FROM clause with no JOIN criteria. Also don't forget to calculate the contract price by using the price per day from the Entertainers table with a 15 percent markup. This technique works because there is only one customer named Matt Berg, only one agent named Karen Smith, and only one entertainment group called Jazz Persuasion. If there happens to be more than one agent or customer with these names, you'll get more than one row inserted into the Engagements table.

Translation/ Clean Up

```
Insert into ~~the~~ engagements ~~table~~
~~into the~~ (customer ID, entertainer ID, agent ID,
start date, end date,
start time, end time, ~~and~~ contract price) ~~columns~~
~~the~~ select~~ion of~~ customer ID, entertainer ID, agent ID,
~~and the values August 15, 2008~~ '2008-08-15',
~~August 16, 2008~~ '2008-08-16',
'~~07:00:00 P.M.~~' '19:00:00', '~~11:00:00 P.M.~~' '23:00:00',
~~and the~~ (entertainer price per day ~~times~~ * 2 ~~times~~ * 1.15)
from ~~the~~ customers, entertainers, ~~and~~ agents ~~tables~~
where ~~the~~ customer first name ~~is~~ = 'Matt'
and ~~the~~ customer last name ~~is~~ = 'Berg'
and ~~the~~ entertainer stage name ~~is~~ = 'Jazz Persuasion'
and ~~the~~ agent first name ~~is~~ = 'Karen'
and ~~the~~ agent last name ~~is~~ = 'Smith'
```

```
SQL         INSERT INTO Engagements
               (CustomerID, EntertainerID, AgentID,
                StartDate, EndDate,
                StartTime, StopTime,
                ContractPrice)
               SELECT Customers.CustomerID,
                  Entertainers.EntertainerID, Agents.AgentID,
                  '2008-08-15', '2008-08-16',
                  '19:00:00', '23:00:00',
                  ROUND(EntPricePerDay * 2 * 1.15, 0)
               FROM Customers, Entertainers, Agents
               WHERE (Customers.CustFirstName = 'Matt')
                AND (Customers.CustLastName = 'Berg')
                AND (Entertainers.EntStageName = 'Jazz
                Persuasion')
                AND (Agents.AgtFirstName = 'Karen')
                AND (Agents.AgtLastName = 'Smith')
```

Row Inserted into the Engagements Table by the CH16_Add_Engagement Query (1 row added)

Customer ID	Entertainer ID	Agent ID	Start Date	End Date	Start Time	Stop Time	Contract Price
10006	1005	4	08/15/2007	08/16/2007	19:00:00	23:00:00	$288.00

School Scheduling Database

"Create a new subject category called 'Italian' with a subject code of 'ITA' in the Humanities department."

❖ **Note** You need the department ID for the Humanities department, so the solution requires a SELECT expression using the Departments table.

Translation/ Clean Up

Insert into ~~the~~ categories ~~table~~
~~the~~ select~~ion of~~ 'ITA' as ~~the~~ category ID,
'Italian' as ~~the~~ category description,
~~and~~ department ID
from ~~the~~ departments ~~table~~
where department name ~~is~~ = 'Humanities'

SQL

```
INSERT INTO Categories
   SELECT 'ITA' AS CategoryID,
      'Italian' AS CategoryDescription,
      Departments.DepartmentID
   FROM Departments
   WHERE Departments.DeptName = 'Humanities'
```

Row Inserted into the Categories Table by the CH16_Add_Category Query (1 row added)

CategoryID	CategoryDescription	DepartmentID
ITA	Italian	3

"Add a new class for subject ID 4 (Intermediate Accounting) to be taught in classroom 3315 for 5 credits starting at 3:00 P.M. for 80 minutes on Tuesdays and Thursdays."

❖ **Note** You can assume that the default value for all schedule days is zero or false, so you need to include a true or 1 value only for Tuesday and Thursday.

Translation/ Clean Up: Insert into ~~the~~ classes ~~table~~ ~~into the columns~~ (subject ID, classroom ID, credits, start time, duration, Tuesday schedule, ~~and~~ Thursday schedule) ~~the~~ values (4, 3315, 5, ~~3 PM~~ 15:00:00, 80, 1, ~~and~~ 1)

SQL:

```
INSERT INTO Classes
     (SubjectID, ClassRoomID, Credits, StartTime,
     Duration, TuesdaySchedule, ThursdaySchedule)
   VALUES (4, 3315, 5, '15:00:00',
      80, 1, 1)
```

Row Inserted into the Classes Table by the CH16_Add_New_Accounting_Class Query (1 row added)

Subject ID	ClassRoom ID	Credits	Start Time	Duration	Tuesday Schedule	Thursday Schedule
4	3315	5	15:00:00	80	1	1

Bowling League Database

"Create a new bowler record for Matthew Patterson by copying the record for Neil Patterson."

❖ **Note** Be sure to set the total pins, games bowled, current average, and current handicap columns to zero.

Translation/ Clean Up

Insert into ~~the~~ bowlers ~~table~~
~~into the columns~~ (bowler last name, bowler first name,
bowler address, bowler city,
bowler state, bowler zip,
bowler phone number, team ID,
bowler total pins, bowler games bowled,
bowler current average, ~~and~~ bowler current handicap)
~~the~~ select~~ion of~~ bowler last name, ~~the value~~ 'Matthew',
bowler address, bowler city,
bowler state, bowler zip,
bowler phone number, team ID,
~~and the values~~ 0, 0,
0, ~~and~~ 0
from ~~the~~ bowlers ~~table~~
where ~~the~~ bowler last name ~~is~~ = 'Patterson'
and ~~the~~ bowler first name ~~is~~ = 'Neil'

SQL

```
INSERT INTO Bowlers
   (BowlerLastName, BowlerFirstName,
    BowlerAddress, BowlerCity,
    BowlerState, BowlerZip,
    BowlerPhoneNumber, TeamID,
    BowlerTotalPins, BowlerGamesBowled,
     BowlerCurrentAverage, BowlerCurrentHcp)
   SELECT Bowlers.BowlerLastName, 'Matthew',
      Bowlers.BowlerAddress, Bowlers.BowlerCity,
      Bowlers.BowlerState, Bowlers.BowlerZip,
      Bowlers.BowlerPhoneNumber, Bowlers.TeamID,
      0, 0,
      0, 0
   FROM Bowlers
   WHERE (Bowlers.BowlerLastName = 'Patterson')
      AND (Bowlers.BowlerFirstName = 'Neil')
```

Row Inserted into the Bowlers Table by the CH16_Add_Bowler Query (1 row added)

Bowler Last Name	Bowler First Name	Bowler Address	Bowler City	Bowler State	Bowler Zip
Patterson	Matthew	16 Maple Lane	Auburn	WA	98002
Bowler Phone Number	**Team ID**	**Bowler Total Pins**	**Bowler Games Bowled**	**Bowler Current Average**	**Bowler Current Hcp**
(206) 555-3487	2	0	0	0	0

"Add a new team called the 'Aardvarks' with no captain assigned."

Translation/ Clean Up: Insert into ~~the~~ teams ~~table into the columns~~ (team name, ~~and~~ captain ID) ~~the~~ values ('Aardvarks', ~~and~~ Null)

SQL:

```
INSERT INTO Teams
    (TeamName, CaptainID)
VALUES ('Aardvarks', NULL)
```

Row Inserted into the Teams Table by the CH16_Add_Team Query (1 row added)

TeamName	CaptainID
Aardvarks	NULL

SUMMARY

We started the chapter with a brief discussion about the INSERT statement used to add data in tables. We introduced the syntax of the INSERT statement and explained a simple example of adding one row using a values list.

Next we discussed the features in most database systems that allow you to generate the next unique value in a table to use as the primary key value for

new rows. We explained that Microsoft SQL Server provides an Identity data type, Microsoft Access provides an AutoNumber data type, and MySQL has an AUTO_INCREMENT attribute for this purpose. We briefly explained the use of the Sequence pseudo-column in the Oracle database system. And finally, we explained how to use a subquery in a VALUES clause to obtain the previous maximum value and add 1.

We explored using a SELECT expression in your INSERT statements to copy one or more rows. First, we reviewed the syntax of the SELECT expression. Next, we showed you how to copy one row from one table to another. We explored copying multiple rows using an example to copy old records to history archive tables. Finally, we showed you how a SELECT expression is often useful for fetching one or more values from a related table to create values to add to your table. The rest of the chapter provided examples of how to build UPDATE queries.

The following section presents a number of problems that you can work out on your own.

Problems for You to Solve

Below, we show you the request statement and the name of the solution query in the sample databases. If you want some practice, you can work out the SQL you need for each request and then check your answer with the query we saved in the samples. Don't worry if your syntax doesn't exactly match the syntax of the queries we saved—as long as your result is the same.

Sales Orders Database

1. *"Customer Liz Keyser wants to order again the products ordered on December 11, 2007. Use June 12, 2008, as the order date and June 15, 2008, as the shipped date."*
   (Hint: You need to copy rows in both the Orders and Order_Details tables. Assume that you can add 1000 to the OrderID column value that you find for the December 11 order for Liz Keyser to generate the new order number.)
   You can find the solution in CH16_Copy_Dec11_Order_For_Keyser (1 row added) and CH16_Copy_Dec11_OrderDetails_For_Keyser (4 rows added).
2. *"Create a new customer record for Mary Baker at 7834 W 32nd Ct., Bothell, WA, 98011, with area code 425 and phone number 555-9876."*
   You can find the solution in CH16_Add_Customer (1 row added).

3. *"Customer Angel Kennedy wants to order again all the products ordered during the month of November 2007. Use June 15, 2008, as the order date and June 18, 2008, as the shipped date."*
   (Hint: You need to copy rows in both the Orders and Order_Details tables. Assume that you can add 1000 to the OrderID column value that you find for the November orders for Angel Kennedy to generate the new order number.)
   You can find the solution in CH16_Copy_November_Orders_For_AKennedy (7 rows added) and CH16_Copy_November_OrderDetails_For_AKennedy (37 rows added).

Entertainment Agency Database

1. *"Agent Marianne Wier would like to book some entertainers, so create a new customer record by copying relevant fields from the Agents table."*
   (Hint: Simply copy the relevant columns from the Agents table to the Customers table.)
   You can find the solution in CH16_Copy_Agent_To_Customer (1 row added).
2. *"Add 'New Age' to the list of musical styles."*
   You can find the solution in CH16_Add_Style (1 row added).
3. *"Customer Doris Hartwig would like to rebook the entertainers she hired to play on December 1, 2007, for August 4, 2008."*
   (Hint: Use a SELECT expression that joins the Customers and Engagements tables, and provide the new engagement dates as literal values.)
   You can find the solution in CH16_Duplicate_Engagement (1 row added).

School Scheduling Database

1. *"Angel Kennedy wants to register as a student. Her husband, John, is already enrolled. Create a new student record for Angel using the information from John's record."*
   You can find the solution in CH16_Add_Student (1 row added).
2. *"Staff member Tim Smith wants to enroll as a student. Create a new student record from Tim's staff record."*
   You can find the solution in CH16_Enroll_Staff (1 row added).

Bowling League Database

1. *"Archive the tournament, tourney match, match game, and bowler scores for all matches played in 2007."*
   (Hint: You need to write four queries to archive rows in the Tournaments, Tourney_Matches, Match_Games, and Bowler_Scores tables.)
   You can find the solution in CH16_Archive_2007_Tournaments_1 (14 rows

added), CH16_Archive_2007_Tournaments_2 (57 rows added), CH16_Archive_2007_Tournaments_3 (168 rows added), and CH16_Archive_2007_Tournaments_4 (1,344 rows added).

2. *"Duplicate all the tournaments and tourney matches played in 2007 for the same week in 2009."*
   (Hint: Assume that you can add 25 to the TourneyID column value for the 2007 tournaments to generate the new tournament ID. You'll need to copy rows in both the Tournaments and the Tourney_Matches tables.)
   You can find the solution in CH16_Copy_2007_Tournaments_1 (14 rows added) and CH16_Copy_2007_Tournaments_2 (57 rows added).

This page intentionally left blank

17

Deleting Sets of Data

"I came to love my rows, my beans,
though so many more than I wanted."
—Henry David Thoreau

Topics Covered in This Chapter

What Is a DELETE?

The DELETE Statement

Uses for DELETE

Sample Statements

Summary

Problems for You to Solve

Now you know how to change data by using an UPDATE statement. You also have learned how to add data by using an INSERT statement. But what about getting rid of unwanted data? For that, you need to use what is arguably the simplest but also the most dangerous statement in SQL—DELETE.

What Is a DELETE?

You learned in the previous chapter that adding data to your tables is fairly straightforward. You can add one row at a time by using a VALUES clause, or you can copy multiple rows by using a SELECT expression. But what do you do if you added a row in error? How do you remove rows you've copied to archive tables? How do you delete a customer who isn't sending you any orders? How do you remove a student who applied for admission but then didn't sign up for any classes? If you want to start over with empty tables, how do you remove all the rows? The answer to all these questions is this: Use a DELETE statement.

Just like all the other statements in SQL, a DELETE statement works with sets of rows. As you'll learn in this chapter, the simplest DELETE statement removes all the rows from the table you specify. But most of the time you'll want to specify the subset of rows to delete. If you guessed that you add a WHERE clause to do that, you're absolutely correct.

❖ **Note** You can find all the sample statements and solutions in the "modify" version of the respective sample databases—SalesOrdersModify, EntertainmentAgencyModify, SchoolSchedulingModify, and BowlingLeagueModify.

The DELETE Statement

The DELETE statement has only three keywords: DELETE, FROM, and WHERE. You can see the diagram of the DELETE statement in Figure 17–1.

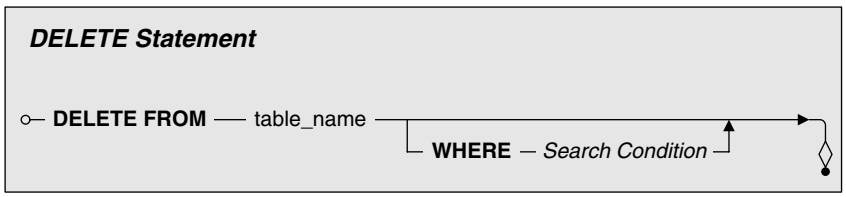


Figure 17–1 *The syntax diagram of the DELETE statement*

We said that the DELETE statement is perhaps the simplest statement in SQL, and we weren't kidding! But it's also the most dangerous statement that you can execute. If you do not include a WHERE clause, the statement removes all the rows in the table you specify. This can be useful when you're testing a new application, for example, so you can empty all the rows from existing tables but keep the table structure. You might also design an application that has working or temporary tables that you load with data to perform a specific task. For example, it's common to use an INSERT statement to copy rows from a very complex SELECT expression into a table that you subsequently use for several static reports. A DELETE statement with no WHERE clause is handy in this case to clean out the old rows before running a new set of reports.

❖ **Note** The SQL Standard indicates that table_name can also be a query (or view) name, but the table implied by the query name must be "updatable." Many database systems support deleting rows from views, and each database system has its own rules about what constitutes an updatable view.

Some database systems also support defining the view (a derived table in SQL Standard terminology) using JOIN and ON keywords in place of table_name. In systems that support using a derived table, you must also specify which table in the JOIN is the target of the delete immediately after the FROM keyword in the form table_name.*. Consult your database system documentation for details. In this chapter, we'll exclusively use a single table as the target for each DELETE statement.

Deleting All Rows

Deleting all rows is almost too easy. Let's construct a DELETE statement using the Bowlers table in the Bowling League sample database.

❖ **Note** Throughout this chapter, we use the "Request/Translation/Clean Up/SQL" technique introduced in Chapter 4, Creating a Simple Query.

"Delete all bowlers."

Translation	Delete all rows from the bowlers table
Clean Up	Delete ~~all rows~~ from ~~the~~ bowlers ~~table~~
SQL	`DELETE` `FROM Bowlers`

If you execute this statement in the sample database, will it actually delete all rows? Actually, no, because we defined a constraint (a referential integrity rule as discussed in Chapter 2, Ensuring Your Database Structure Is Sound) between the Bowlers table and the Bowler_Scores table. If any rows exist for a particular bowler in the Bowler_Scores table, your database system should not allow you to delete the row in the Bowlers table for that bowler.

Two bowlers in the modify version of the Bowling League sample database do not have any scores, so you should be able to delete those records with this simple DELETE statement. Even if you really didn't mean to delete any rows at all, those two rows will be gone forever. Well, maybe. First, many database systems maintain a log of changes you make to tables. It is sometimes possible to recover lost data from the system logs. Remember also our brief discussion about transactions in Chapter 15, Updating Sets of Data. If you start a transaction (or the system starts one for you), you can roll back any pending changes if you encounter any errors.

You might also remember that we told you that Microsoft Office Access is one database system that automatically starts a transaction for you whenever you execute a query from the user program interface. If you try to run this query in Microsoft Access, it will first prompt you with a warning about how many rows are about to be deleted. You can cancel the delete at that point when you realize that the database is about to attempt to delete all the rows in the table. If you let the system continue beyond the first warning, you'll receive the error dialog box shown in Figure 17–2.

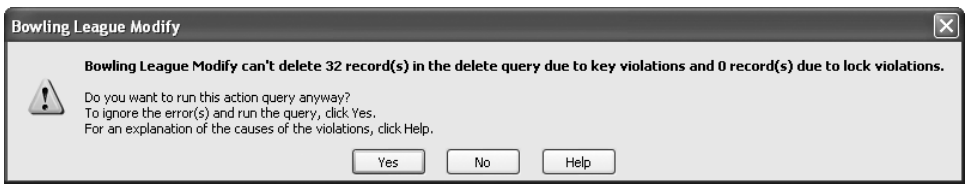

Bowling League Modify
Bowling League Modify can't delete 32 record(s) in the delete query due to key violations and 0 record(s) due to lock violations.
Do you want to run this action query anyway?
To ignore the error(s) and run the query, click Yes.
For an explanation of the causes of the violations, click Help.
Yes
No
Help

Figure 17–2 *Some database systems warn you if executing a DELETE statement will cause errors.*

You can see that 32 of the 34 records in the table won't be deleted because of "key violations." This is an obtuse way to tell you: "Hey, dummy, you've still got rows in the Bowler_Scores table for some of these bowlers you're trying to delete." Click No at this point, and the database system will execute a ROLLBACK on your behalf—none of the rows will be deleted. Click Yes, and the database system executes a COMMIT to permanently delete the two rows for bowlers who have no scores.

In the Sample Statements section later in this chapter, we'll show you two ways to safely delete bowlers who haven't bowled any games if that's what you really want to do.

Deleting Some Rows

Most of the time, you'll want to limit the rows that you delete. You can do that by adding a WHERE clause to specifically filter the rows to be deleted. Your WHERE clause can be as simple or as complex as any you've learned about for SELECT or UPDATE statements.

Using a Simple WHERE Clause

Let's start with something simple. Suppose you want to delete in the Sales Orders database any orders that have a zero order total. Your request might look like this.

"Delete orders that have a zero order total."

Translation	Delete from the orders table where the order total is zero
Clean Up	Delete from ~~the~~ orders ~~table~~ where ~~the~~ order total ~~is~~ = ~~zero~~ 0
SQL	`DELETE FROM Orders` `WHERE OrderTotal = 0`

The WHERE clause uses a simple comparison predicate to find only the rows that have an order total equal to zero. If you execute this query in the sample database, you'll find that it deletes 11 rows. You can find this query saved as CH17_Delete_Zero_OrdersA.

Safety First: Ensuring That You're Deleting the Correct Rows

Even for simple DELETE queries, we strongly recommend that you verify that you'll be deleting the correct rows. How do you do that? As we mentioned, most of the time you'll add a WHERE clause to select a subset of rows to delete. Why not build a SELECT query first to return the rows that you intend to remove?

"List all the columns from the Orders table for the orders that have a zero order total."

Translation	Select all columns from the orders table where the order total is zero
Clean Up	Select ~~all columns~~ * from ~~the~~ orders ~~table~~ where ~~the~~ order total ~~is~~ = ~~zero~~ 0

```
SQL         SELECT *
            FROM Orders
            WHERE OrderTotal = 0
```

If you run this query on the Sales Orders sample database, your result should look like Figure 17–3.

OrderNumber	OrderDate	ShipDate	CustomerID	EmployeeID	OrderTotal
198	2007-10-07	2007-10-09	1002	703	$0.00
216	2007-10-11	2007-10-11	1016	707	$0.00
305	2007-10-31	2007-11-04	1013	708	$0.00
361	2007-11-11	2007-11-12	1016	706	$0.00
484	2007-12-08	2007-12-09	1021	707	$0.00
523	2007-12-14	2007-12-16	1003	704	$0.00
629	2008-01-07	2008-01-11	1014	704	$0.00
632	2008-01-07	2008-01-11	1001	706	$0.00
689	2008-01-14	2008-01-15	1015	705	$0.00
753	2008-01-27	2008-01-29	1013	701	$0.00
816	2008-02-08	2008-02-11	1011	701	$0.00

Figure 17–3 *Verifying the rows you want to delete*

Note that we used the shortcut * character to indicate we wanted to see all columns. If the result set shows all the rows you want to delete, you can transform your SELECT statement into the correct DELETE statement by simply replacing SELECT * with DELETE. Figure 17–4 shows how to transform this SELECT statement into the correct DELETE syntax.

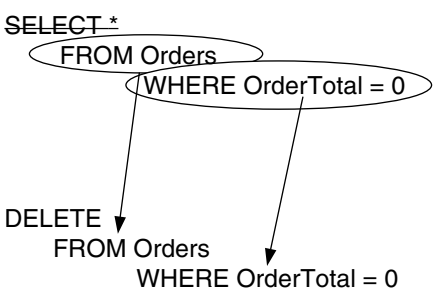


Figure 17–4 *Converting a verifying SELECT query into a DELETE statement*

This conversion is so simple that it would be silly to not create the SELECT statement first to make sure you're deleting the rows you want. Remember, unless you've protected your DELETE inside a transaction, after you execute a DELETE statement, the rows are gone for good.

Using a Subquery

The query explained in the previous section to delete all orders that have a zero order total seems simple enough. But keep in mind that the OrderTotal column is a calculated value. (We showed you how to calculate and set the total using an UPDATE query in Chapter 15.) What if the user or the application failed to run the update after adding, changing, or deleting one or more order detail rows? Your simple query might attempt to delete an order that still has rows in the Order_Details table, and it might miss some orders that had all the order details removed but didn't have the total updated.

A safer way to ensure that you're deleting orders that have no order details is to use a subquery to check for matching rows in the Order_Details table. Your request might look like this.

"Delete all orders that have no items ordered."

Translation	Delete the rows from the orders table where the order number is not in the selection of the order number from the order details table
Clean Up	Delete ~~the rows~~ from ~~the~~ orders ~~table~~ where ~~the~~ order number ~~is~~ not in ~~the~~ (select~~ion of the~~ order number from ~~the~~ order details) ~~table~~

SQL	`DELETE FROM Orders`
	`WHERE OrderNumber NOT IN`
	`(SELECT OrderNumber`
	`FROM Order_Details)`

That's a bit more complex than the simple comparison for an order total equal to zero, but it ensures that you delete only orders that have no matching rows in the Order_Details table. You can find this query saved as CH17_Delete_Zero_OrdersB in the sample database. This more complex query might actually find and delete some rows that have a nonzero order total that wasn't correctly updated when the last order item was deleted.

To construct the WHERE clause for DELETE queries, you'll probably use IN, NOT IN, EXISTS, or NOT EXISTS quite frequently. (Reread Chapter 11, Subqueries, if you need a refresher.) Let's look at one more example that requires a complex WHERE clause to filter the rows to be deleted.

"Delete all orders and order details for orders placed before January 1, 2008, that have been copied to the archive tables."

Remember that in Chapter 16, Inserting Sets of Data, we showed you how to use INSERT to copy a set of old rows to one or more archive tables. After you copy the rows, you can often make the processing in the main part of your application more efficient by deleting the rows that you have archived. As implied by the request, you need to delete rows from two tables, so let's break it down into two requests. You need to delete from the Order_Details table first because a defined referential integrity rule won't let you delete rows in the Orders table if matching rows exist in the Order_Details table.

"Delete all order details for orders placed before January 1, 2008, that have been copied to the archive table."

Do you see a potential danger here? One way to solve the problem would be to simply delete rows from orders that were placed before January 1, 2008.

Translation
Delete rows from the order details table
where the order number is in
the selection of the order number
from the orders table
where the order date
is earlier than January 1, 2008

Clean Up
Delete ~~rows~~ from ~~the~~ order details ~~table~~
where ~~the~~ order number ~~is~~ in
~~the~~ (select~~ion of the~~ order number
from ~~the~~ orders ~~table~~
where ~~the~~ order date
~~is earlier than~~ < ~~January 1, 2008~~ '2008-01-01')

SQL

```
DELETE FROM Order_Details
WHERE OrderNumber IN
(SELECT OrderNumber
FROM Orders
WHERE OrderDate
< '2008-01-01')
```

You can find this query saved as CH17_Delete_Archived_Order_Details_Unsafe. What if someone else promised to run the INSERT query to archive the rows but really didn't? If you run this query, you'll delete all the order details for orders placed before January 1, 2008, regardless of whether the

rows actually exist in the archive table. A safer way is to delete only the rows that you first verify are in the archive table. Let's try again.

Translation	Delete rows from the order details table where the order number is in the selection of order number from the order details archive table
Clean Up	Delete ~~rows~~ from ~~the~~ order details ~~table~~ where ~~the~~ order number ~~is~~ in ~~the~~ (select~~ion of~~ order number from ~~the~~ order details archive) ~~table~~

SQL	`DELETE FROM OrderDetails`
	`WHERE OrderNumber IN`
	`(SELECT OrderNumber`
	`FROM Order_Details_Archive)`

You can find this query saved as CH17_Delete_Archived_Order_Details_OK. Notice that the query doesn't care at all about the order date. However, it is much safer because it is deleting only the rows in the main table that have a matching order number in the archive table. If you want to be sure you're deleting rows from orders that are before January 1, 2008, and that are already in the archive table, you can use both IN predicates in your query combined with the AND Boolean operator.

Uses for DELETE

At this point, you should have a good understanding of how to delete one or more rows in a table—either all the rows or a selection of rows determined by using a WHERE clause. The best way to give you an idea of the wide range of uses for the DELETE statement is to list some problems you can solve with this statement and then present a set of examples in the Sample Statements section. Here's just a small list of the types of problems that you can solve with DELETE.

"Delete products that have never been ordered."

"Delete all entertainers who have never been hired."

"Delete bowlers who have not bowled any games."

"Delete all students who are not registered for any class."

"Delete any categories that have no products."

"Delete customers who have never booked an entertainer."

"Delete teams that have no bowlers assigned."
"Delete all classes that have never had a student registered."
"Delete customers who haven't placed an order."
"Delete musical styles that aren't played by any entertainer."
"Delete all bowling matches that have not been played."
"Delete subjects that have no classes."
"Delete all engagements that have been copied to the archive table."
"Delete all the tournament data that has been copied to the archive tables."
"Delete vendors who do not provide any products."
"Delete members who are not part of an entertainment group."
"Delete employees who haven't sold anything."

Sample Statements

You now know the mechanics of constructing DELETE queries. Let's take a look at a set of samples, all of which request that one or more rows be deleted from a table. These examples come from four of the sample databases.

We've also included a view of the data that the sample DELETE statement should remove from the target table and a count of the number of rows that should be deleted. The name that appears immediately before the count of rows deleted is the name we gave each query in the sample data on the companion CD you'll find bound into the back of the book. We stored each query in the appropriate sample database (as indicated within the example) and prefixed the names of the queries relevant to this chapter with "CH17." You can follow the instructions in the Introduction of this book to load the samples onto your computer and try them.

❖ **Note** Remember that all the column names and table names used in these examples are drawn from the sample database structures shown in Appendix B, Schema for the Sample Databases. To simplify the process, we have combined the Translation and Clean Up steps for all the examples. These samples assume you have thoroughly studied and understood the concepts covered in previous chapters.

Sales Orders Database

"Delete customers who haven't placed an order."

Translation/ Clean Up: Delete ~~rows~~ from ~~the~~ customers ~~table~~ where ~~the~~ customer ID ~~is~~ not in ~~the~~ (select~~ion of the~~ customer ID from ~~the~~ orders) ~~table~~

SQL:

```
DELETE FROM Customers
WHERE CustomerID NOT IN
(SELECT CustomerID
FROM Orders)
```

Row Deleted from the Customers Table by the CH17_Delete_Customers_Never_Ordered Query (1 row deleted)

Customer ID	Cust First Name	Cust Last Name	Cust Street Address	Cust City	Cust State	Cust Zip Code	Cust Area Code	Cust phone Number
1028	Jeffrey	Tirekicker	15622 NE 42nd Ct	Redmond	WA	98052	425	555-9999

"Delete vendors who do not provide any products."

Translation/ Clean Up: Delete ~~rows~~ from ~~the~~ vendors ~~table~~ where ~~the~~ vendor ID ~~is~~ not in ~~the~~ (select~~ion of~~ vendor ID from ~~the~~ product vendors) ~~table~~

SQL:

```
DELETE FROM Vendors
WHERE VendorID NOT IN
(SELECT VendorID
FROM Product_Vendors)
```

Row Deleted from the Vendors Table by the CH17_Delete_Vendors_No_Products Query (1 row deleted)

Vendor ID	VendName	VendStreet Address	Vend City	Vend State	Vend ZipCode	<<other columns>>
11	Astro Paper Products	5639 N. Riverside	Chicago	IL	60637	. . .

Entertainment Agency Database

"Delete all entertainers who have never been hired."

Translation 1/ Clean Up: Delete ~~rows~~ from ~~the~~ entertainer members ~~table~~ where ~~the~~ entertainer ID ~~is~~ not in ~~the~~ (select~~ion of~~ entertainer ID from ~~the~~ engagements) ~~table~~

SQL:

```
DELETE FROM Entertainer_Members
WHERE EntertainerID NOT IN
(SELECT EntertainerID
FROM Engagements)
```

Row Deleted from the Entertainer_Members Table by the CH17_Delete_Entertainers_Not_Booked1 Query (1 row deleted)

EntertainerID	MemberID	Status
1009	121	2

Translation 2/ Clean Up: Delete ~~rows~~ from ~~the~~ entertainers ~~table~~ where ~~the~~ entertainer ID ~~is~~ not in ~~the~~ (select~~ion of~~ entertainer ID from ~~the~~ engagements) ~~table~~

SQL:

```
DELETE FROM Entertainers
WHERE EntertainerID NOT IN
(SELECT EntertainerID
FROM Engagements)
```

Row Deleted from the Entertainers Table by the CH17_Delete_Entertainers_Not_Booked2 Query (1 row deleted)

Entertainer ID	EntStage Name	Ent SSN	EntStreet Address	Ent City	Ent State	Ent ZipCode	<<*other columns*>>
1009	Katherine Ehrlich	888-61-1103	777 Fenexet Blvd	Woodinville	WA	98072	…

"Delete all engagements that have been copied to the archive table."

Translation/ Clean Up: Delete ~~rows~~ from ~~the~~ engagements ~~table~~ where ~~the~~ engagement ID ~~is~~ in ~~the~~ (selection ~~of~~ engagement ID from ~~the~~ engagements archive) ~~table~~

SQL:

```
DELETE FROM Engagements
WHERE EngagementID IN
(SELECT EngagementID
FROM Engagements_Archive)
```

❖ **Note** In order to find rows to delete, you must first run the CH16_Archive_Engagements query to copy data to the archive table. The archive table in the original sample database is empty.

Rows Deleted from the Engagements Table by the CH17_Remove_Archived_Engagements Query (56 rows deleted if you first run CH16_Archive_Engagements)

Engagement Number	Start Date	End Date	Start Time	Stop Time	Contract Price	Customer ID	Agent ID	Entertainer ID
2	2007-09-01	2007-09-05	13:00	15:00	$200.00	10006	4	1004
3	2007-09-10	2007-09-15	13:00	15:00	$590.00	10001	3	1005
4	2007-09-11	2007-09-17	20:00	0:00	$470.00	10007	3	1004
5	2007-09-11	2007-09-14	16:00	19:00	$1,130.00	10006	5	1003
6	2007-09-10	2007-09-14	15:00	21:00	$2,300.00	10014	7	1008
<< more rows here >>								

School Scheduling Database

"Delete all classes that have never had a student registered."

❖ **Note** You need to delete the rows from the Faculty_Classes table first and then delete from the Classes table because the database has an integrity rule that won't let you delete rows in the Classes table when matching rows exist in the Faculty_Classes table.

Translation 1/ Clean Up

Delete from ~~the~~ faculty classes ~~table~~ where ~~the~~ class ID ~~is~~ not in ~~the~~ (select~~ion of~~ class ID from ~~the~~ student schedules) ~~table~~

SQL

```
DELETE FROM Faculty_Classes
WHERE ClassID NOT IN
(SELECT ClassID
FROM Student_Classes)
```

Rows Deleted from the Faculty_Classes Table by the CH17_Delete_Classes_No_Students_1 Query (60 rows deleted)

ClassID	StaffID
1002	98036
1004	98019
1006	98045
1020	98028
1030	98036
1156	98055
1162	98064
1183	98005
1184	98011
<< more rows here >>	

Translation 2/ Clean Up	Delete from ~~the~~ classes ~~table~~ where ~~the~~ class ID ~~is~~ not in ~~the~~ (selection ~~of~~ class ID from ~~the~~ student schedules) ~~table~~

SQL	`DELETE FROM Classes`
	`WHERE ClassID NOT IN`
	`(SELECT ClassID`
	`FROM Student_Schedules)`

Rows Deleted from the Classes Table by the CH17_Delete_Classes_No_Students_2 Query (61 rows deleted)

Class ID	Subject ID	Class RoomID	Credits	Start Time	Duration	Monday Schedule	<<other columns>>
1002	12	1619	4	15:30	110	Yes	…
1004	13	1627	4	8:00	50	Yes	…
1006	13	1627	4	9:00	110	Yes	…
1020	15	3404	4	13:00	110	Yes	…
1030	16	1231	5	11:00	50	Yes	…
1156	37	3443	5	8:00	50	Yes	…
<< more rows here >>							

Bowling League Database

"Delete bowlers who have not bowled any games."

❖ **Note** You can solve this request by deleting bowlers whose number of games bowled is zero or by deleting bowlers who have no rows in the Bowler_Scores table. The second method is safer because it doesn't depend on the calculated value of the games bowled, but let's solve it both ways.

Translation 1/ Clean Up: Delete ~~rows~~ from ~~the~~ bowlers ~~table~~ where ~~the~~ bowler games bowled ~~is~~ = ~~zero~~ 0

SQL:

```
DELETE FROM Bowlers
WHERE BowlerGamesBowled = 0
```

Rows Deleted from the Bowlers Table by the CH17_Delete_Bowlers_No_Games Query (2 rows deleted)

BowlerID	Bowler Last Name	Bowler First Name	<<*other columns*>>	Bowler Games Bowled	Bowler Current Average	Bowler Current Hcp
33	Patterson	Kerry	…	0	0	0
34	Patterson	Maria	…	0	0	0

Translation 2/ Clean Up: Delete ~~rows~~ from ~~the~~ bowlers ~~table~~ where ~~the~~ bowler ID ~~is~~ not in ~~the~~ (select~~ion of~~ bowler ID from ~~the~~ bowler scores) ~~table~~

SQL:

```
DELETE FROM Bowlers
WHERE BowlerID NOT IN
(SELECT BowlerID
FROM Bowler_Scores)
```

Rows Deleted from the Bowlers Table by the CH17_Delete_Bowlers_No_Games_Safe Query (2 rows deleted)

BowlerID	Bowler Last Name	Bowler First Name	*<<other columns>>*	Bowler Games Bowled	Bowler Current Average	Bowler Current Hcp
33	Patterson	Kerry	. . .	0	0	0
34	Patterson	Maria	. . .	0	0	0

"Delete teams that have no bowlers assigned."

Translation/ Clean Up: Delete from ~~the~~ teams ~~table~~ where ~~the~~ team ID ~~is~~ not in ~~the~~ (selecti~~on of~~ team ID from ~~the~~ bowlers) ~~table~~

SQL:

```
DELETE FROM Teams
WHERE TeamID NOT IN
(SELECT TeamID
FROM Bowlers)
```

Rows Deleted from the Bowlers Table by the CH17_Delete_ Teams_No_Bowlers Query (2 rows deleted)

TeamID	TeamName	CaptainID
9	Huckleberrys	7
10	Never Show Ups	22

SUMMARY

We started the chapter with a brief discussion about the DELETE statement used to delete rows from tables. We introduced the syntax of the DELETE statement and explained a simple example of deleting all the rows in a table. We briefly reviewed transactions and showed you how the Microsoft Access database system uses transactions to help protect you from mistakes.

Next we discussed using a WHERE clause to limit the rows you are deleting. We explained how to use a SELECT statement to verify the rows you plan to delete and how to convert the SELECT statement into a DELETE statement.

Finally, we extensively explored using subqueries to test for rows to delete based on the existence or nonexistence of related rows in other tables. The rest of the chapter provided examples of how to build DELETE queries.

The following section presents a number of problems that you can work out on your own.

Problems for You to Solve

Below, we show you the request statement and the name of the solution query in the sample databases. If you want some practice, you can work out the SQL you need for each request and then check your answer with the query we saved in the samples. Don't worry if your syntax doesn't exactly match the syntax of the queries we saved—as long as your result is the same.

Sales Orders Database

1. *"Delete products that have never been ordered."*
   (Hint: You need to delete from the Product_Vendors table first and then from the Products table.)
   You can find the solution in CH17_Delete_Products_Never_Ordered_1 (4 rows deleted) and CH17_Delete_Products_Never_Ordered_2 (2 rows deleted).
2. *"Delete employees who haven't sold anything."*
   You can find the solution in CH17_Delete_Employees_No_Orders (1 row deleted).
3. *"Delete any categories that have no products."*
   You can find the solution in CH17_Delete_Categories_No_Products (1 row deleted).

Entertainment Agency Database

1. *"Delete customers who have never booked an entertainer."*
   You can find the solution in CH17_Delete_Customers_Never_Booked (2 rows deleted).
2. *"Delete musical styles that aren't played by any entertainer."*
   You can find the solution in CH17_Delete_Styles_No_Entertainer (8 rows deleted).
3. *"Delete members who are not part of an entertainment group."*
   You can find the solution in CH17_Delete_Members_Not_In_Group (no rows deleted).

School Scheduling Database

1. *"Delete all students who are not registered for any class."*
   You can find the solution in CH17_Delete_Students_No_Classes (1 row deleted).
2. *"Delete subjects that have no classes."*
   (Hint: You need to delete rows from both the Faculty_Subjects and the Subjects tables.)
   You can find the solution in CH17_Delete_Subjects_No_Classes_1 (6 rows deleted) and CH17_Delete_Subjects_No_Classes_2 (3 rows deleted).

Bowling League Database

1. *"Delete all the tournament data that has been copied to the archive tables."*
   (Hint: You need to delete rows from the Bowler_Scores, Match_Games, Tourney_Matches, and Tournaments tables. You should find no rows to delete unless you have executed the four archive queries from Chapter 16.)
   You can find the solution in CH17_Delete_Archived_2007_Tournaments_1 (1,344 rows deleted), CH17_Delete_Archived_2007_Tournaments_2 (168 rows deleted), CH17_Delete_Archived_2007_Tournaments_3 (57 rows deleted), and CH17_Delete_Archived_2007_Tournaments_4 (14 rows deleted).
2. *"Delete all bowling matches that have not been played."*
   You can find the solution in CH17_Delete_Matches_Not_Played (1 row deleted).

This page intentionally left blank

In Closing

"That is what learning is.
You suddenly understand something
you've understood all your life, but in a new way."
—Doris Lessing

You now have all the tools you need to query or change a database successfully. You've learned how to create both simple and complex SELECT statements and how to work with various types of data. You've also learned how to filter data with search conditions, work with multiple tables using JOINs, and produce statistical information by grouping data. And finally, you learned how to update, add, and delete data in your tables.

As with any new endeavor, there's always more to learn. Your next task is to take the techniques you've learned in this book and apply them within your database system. Be sure to refer to your database system's documentation to determine whether there are any differences between standard SQL syntax and the SQL syntax your database uses. If your database allows you to create queries using a graphical interface, you'll probably find that the interface now makes more sense and is much easier to use.

Also remember that we focused only on the data manipulation portion of SQL—there are still many parts to SQL that you can delve into should you be so inspired. For example, you could learn how to create data structures; incorporate several tables into a single view, function, or stored procedure; or embed SQL statements within an application program. If you want to learn more about SQL, we suggest you start with any of the books we've listed in Appendix D, Suggested Reading.

We hope you've enjoyed reading this book as much as we've enjoyed writing it. We know that books on this subject tend to be rather dry, so we decided to have a little fun and inject some humor wherever we could. There's

absolutely no reason why learning should be boring and tedious. On the contrary, you should look forward to learning something new each day.

Writing a book is always a humbling experience. It makes you realize just how much more there is to learn about the subject at hand. And as you work through the writing process, it is inevitable that you'll see things from a fresh perspective and in a different light. We found out just how much Doris Lessing's statement rings true.

We hope you will, too.

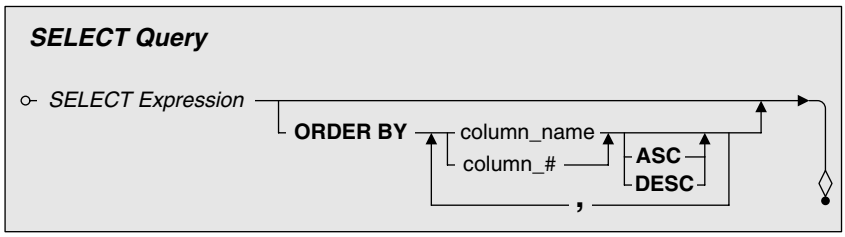
Appendices

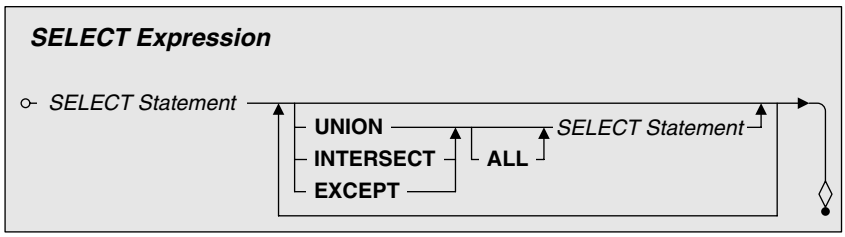
This page intentionally left blank

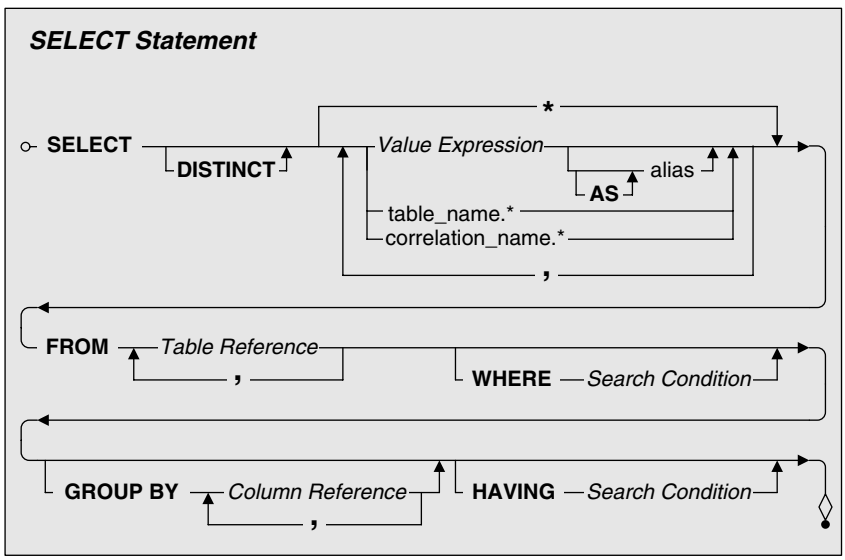
A

SQL Standard Diagrams

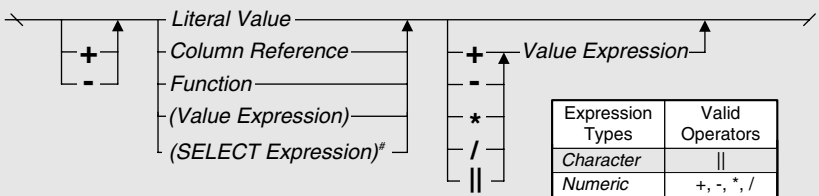
Here are the complete diagrams for all the SQL grammar and syntax we've covered throughout the book.







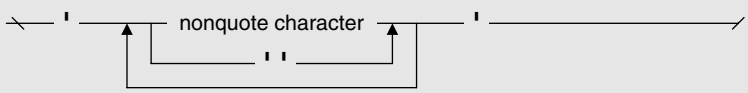
Value Expression



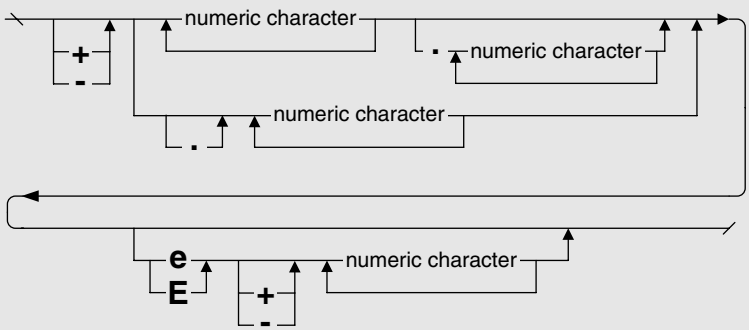
Expression Types	Valid Operators
Character	||
Numeric	+, -, *, /
Date / Time	+, -
Interval	+, -, *, /

Literal Value

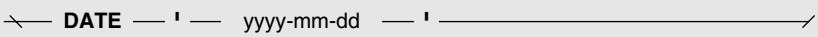
Character String



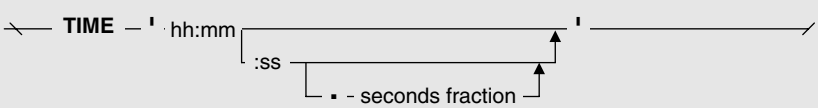
Numeric



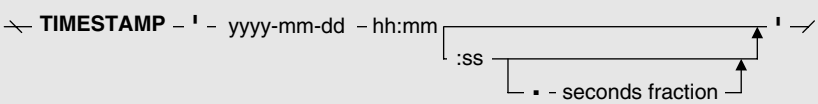
Date



Time

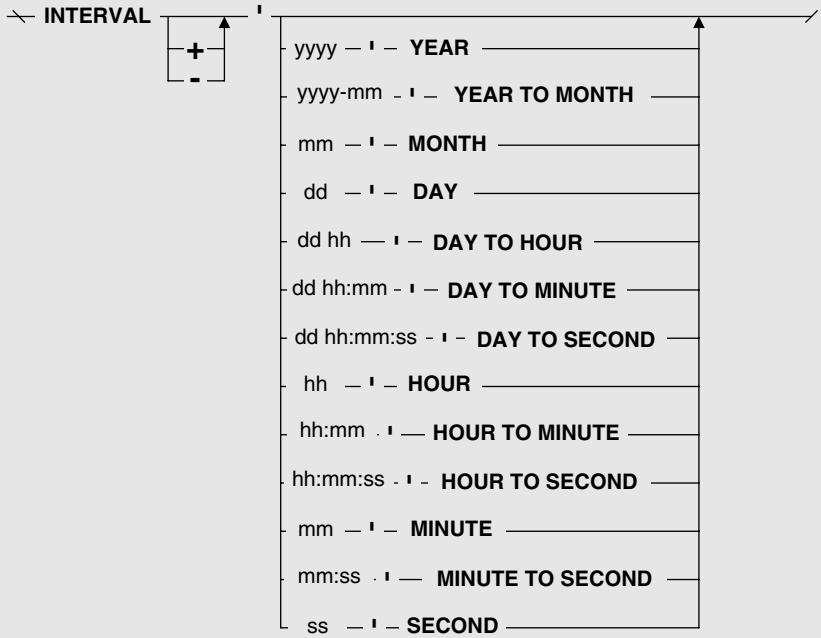


Timestamp

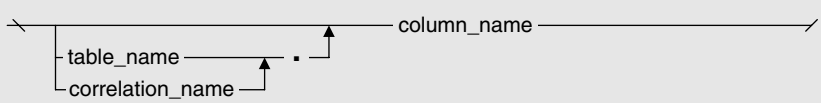


Literal Value (cont.)

Interval

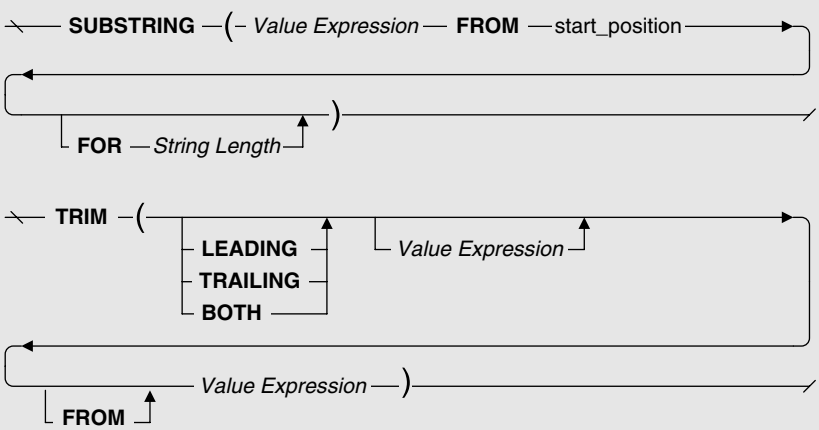


Column Reference

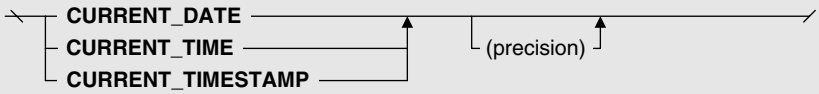


Functions

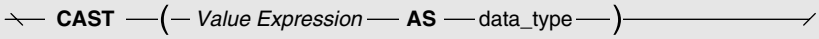
Character String



Datetime

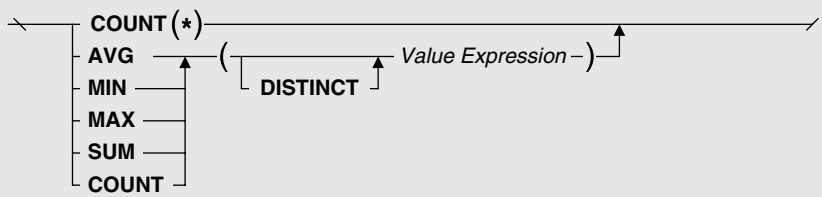


Conversion



Functions (cont.)

Aggregate



Numeric

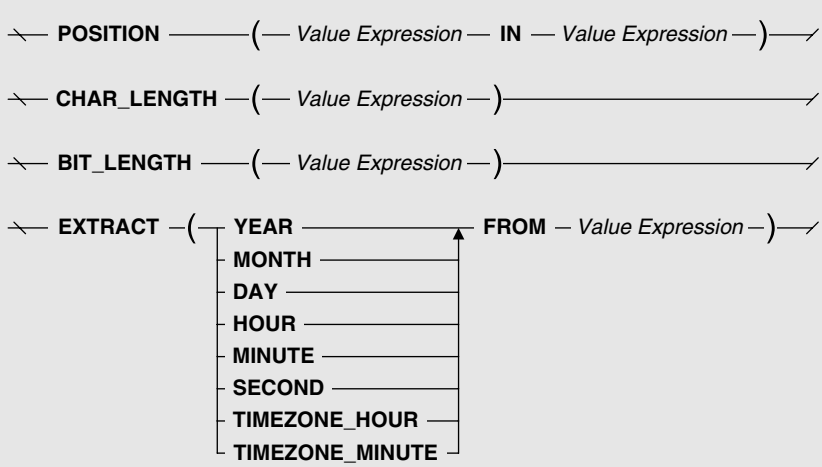
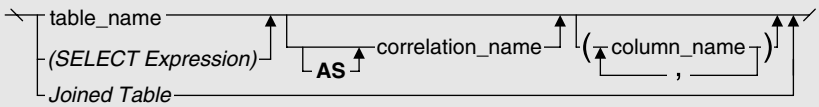
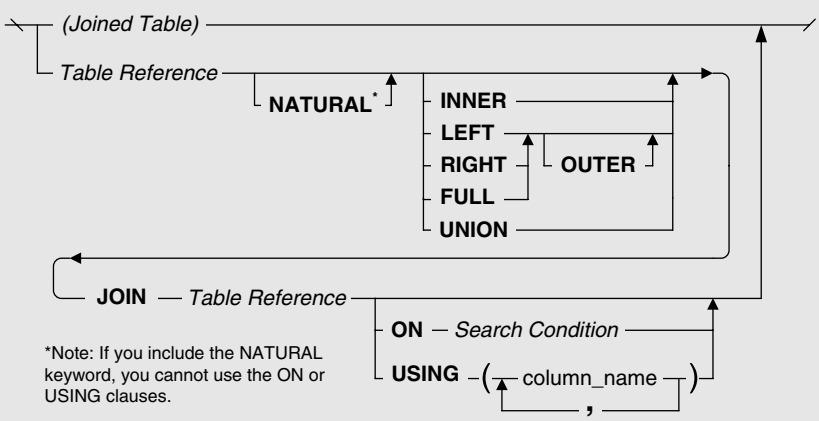


Table Reference

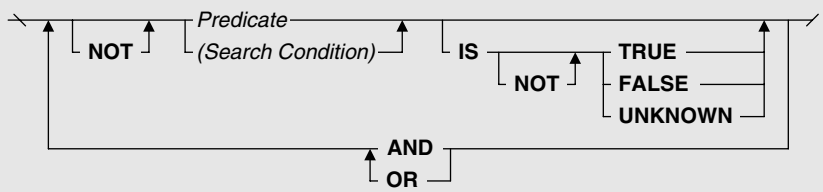


Joined Table



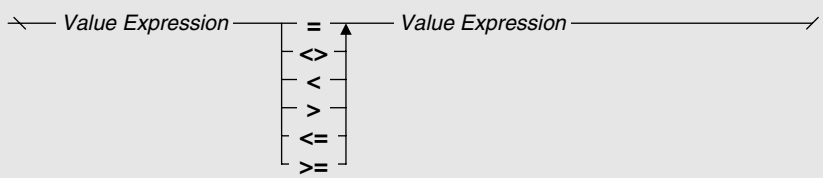
*Note: If you include the NATURAL keyword, you cannot use the ON or USING clauses.

Search Condition

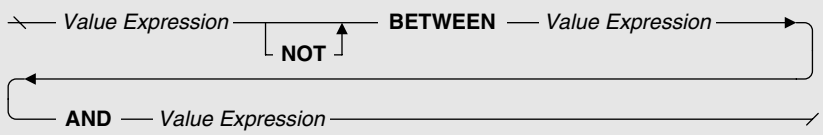


Predicate

Comparison

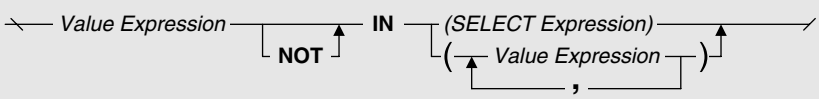


Range

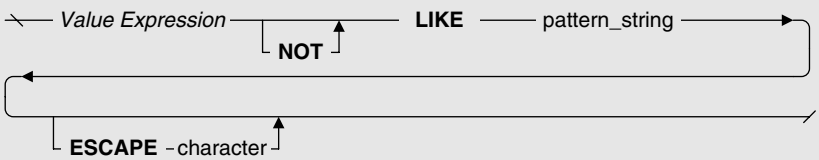


Predicate (cont.)

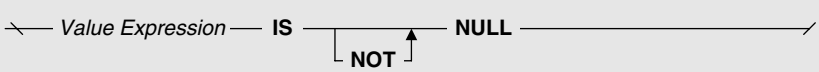
Set Membership



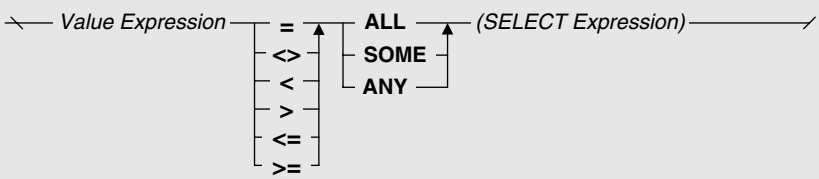
Pattern Match



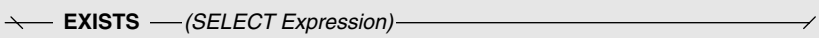
Null



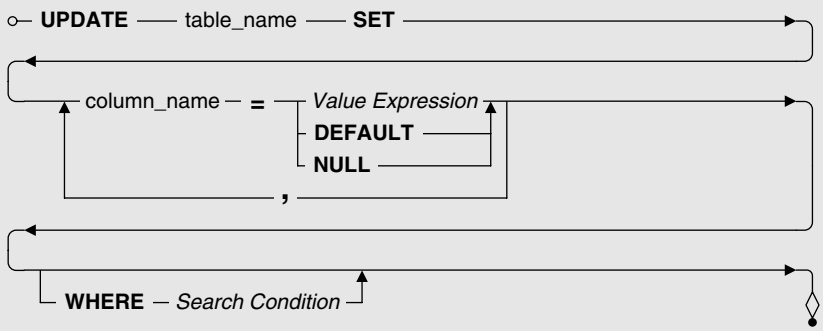
Quantified



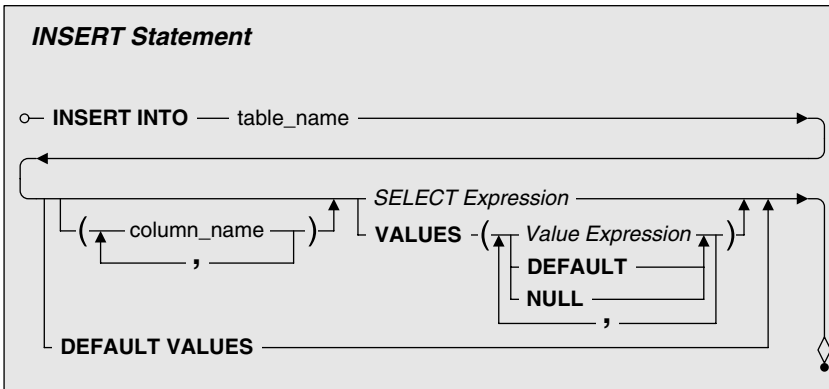
Existence



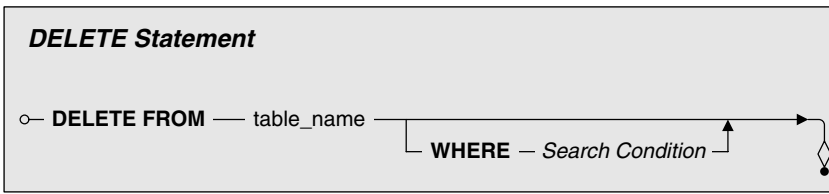
UPDATE Statement



INSERT Statement



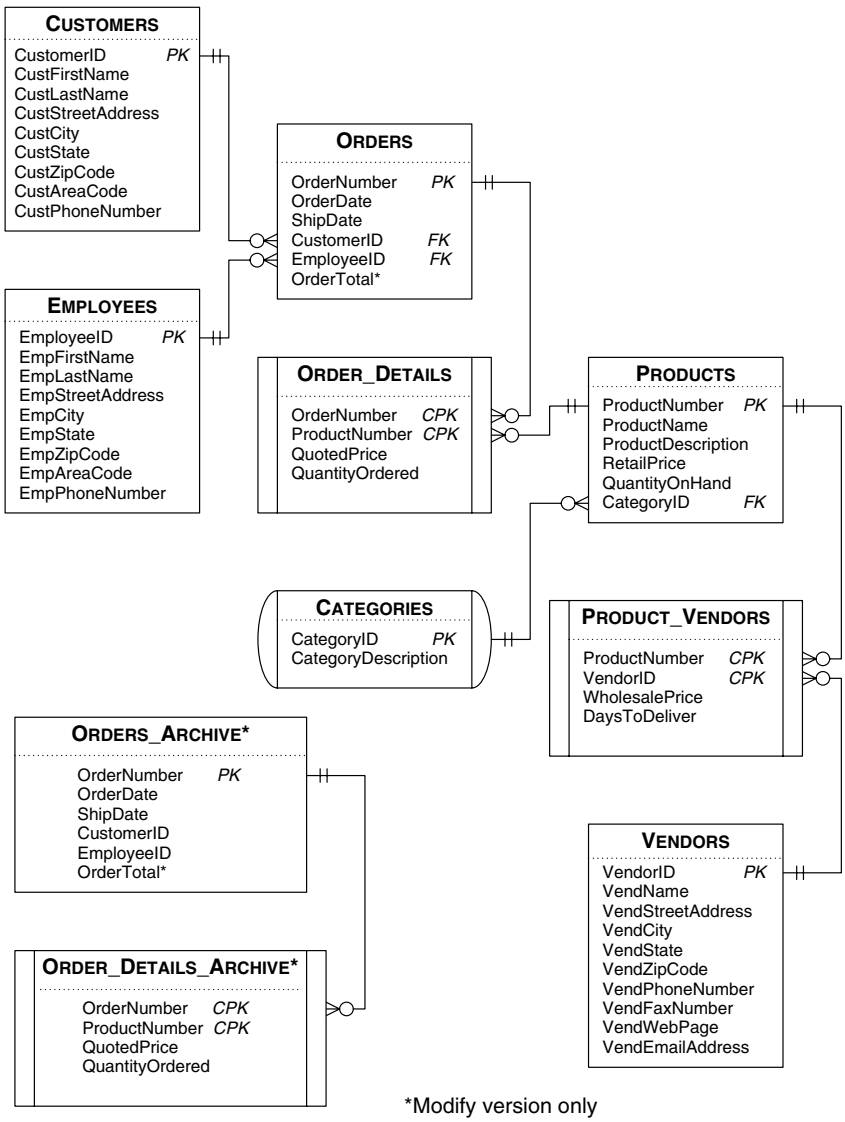
DELETE Statement



This page intentionally left blank

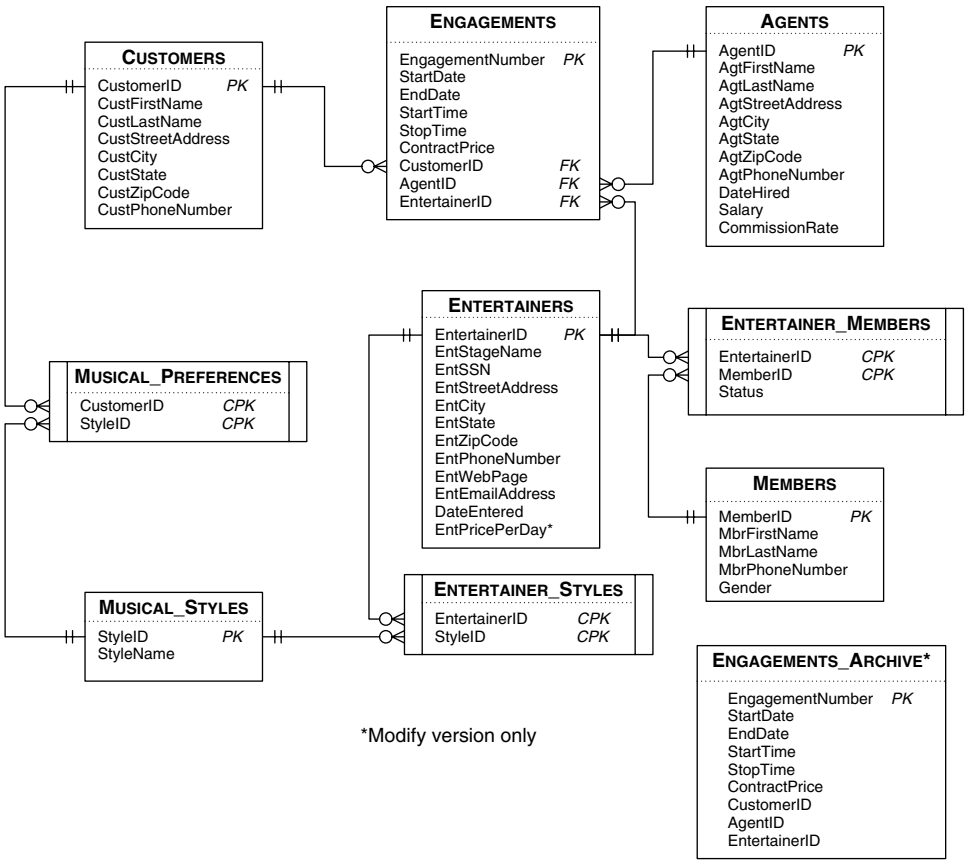
Schema for the Sample Databases

Sales Orders Database



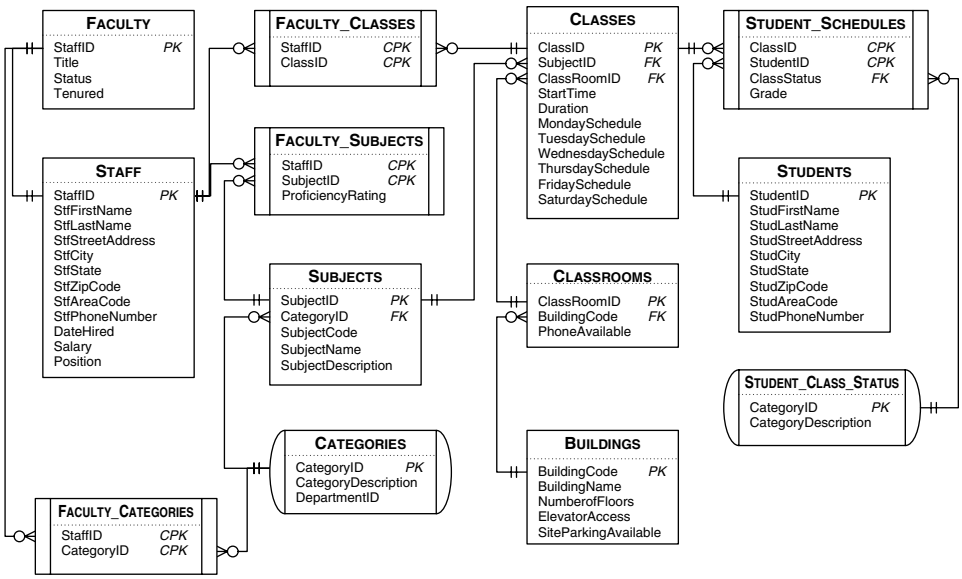
*Modify version only

Entertainment Agency Database

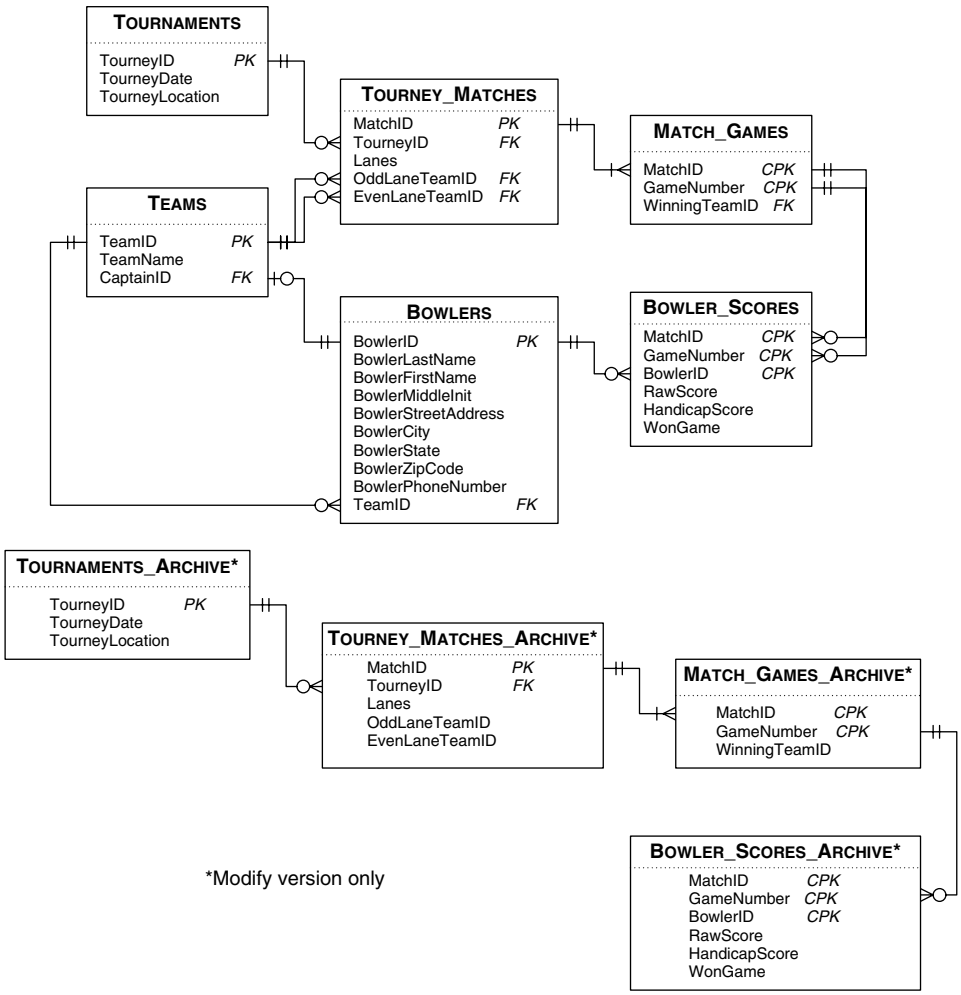


*Modify version only

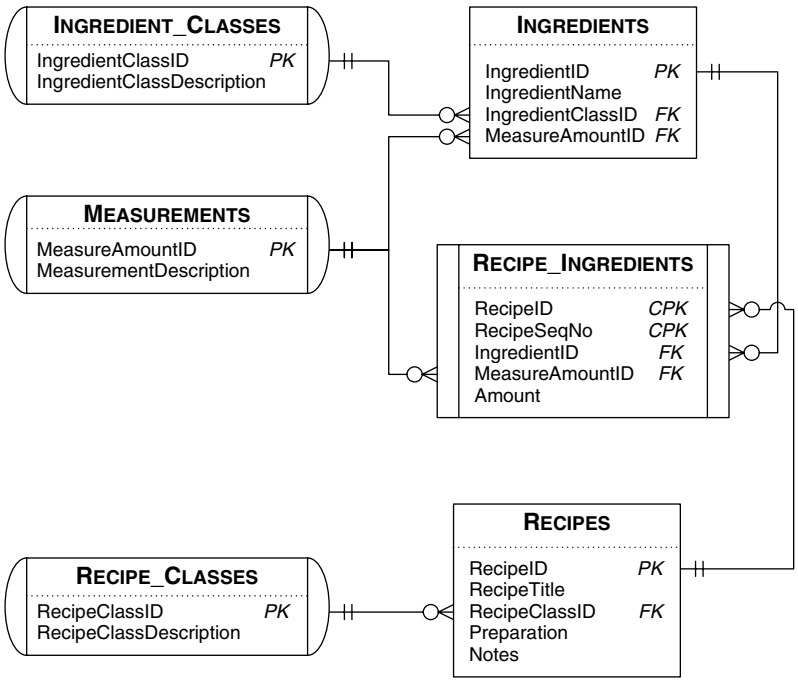
School Scheduling Database



Bowling League Database



Recipes Database



This page intentionally left blank

C

Date and Time Functions

As mentioned in Chapter 5, Getting More Than Simple Columns, each database system has a variety of functions that you can use to fetch or manipulate date and time values. The SQL Standard specifically defines three functions, CURRENT_DATE, CURRENT_TIME, and CURRENT_TIMESTAMP, but not all commercial database systems support all three function calls. To help you work with date and time values in your database system, we've compiled a list of functions for several of the major database systems that you can use to work with date and time values. The lists in this appendix include the function name and a brief description of its use. Consult your database documentation for the specific syntax to use with each function.

IBM DB2

Function Name	Description
CURRENT DATE	Obtains the current date value.
CURRENT TIME	Obtains the current time value in the local time zone.
CURRENT TIMESTAMP	Obtains the current date and time in the local time zone.
DATE(<*expression*>)	Evaluates the expression and returns a date.
DAY(<*expression*>)	Evaluates the expression and returns the day part of a date, timestamp, or date interval.
DAYOFMONTH(<*expression*>)	Evaluates the expression and returns the day part of a date or timestamp.
DAYOFWEEK(<*expression*>)	Evaluates the expression and returns the day number of the week, where 1 = Sunday.
DAYOFWEEK_ISO(<*expression*>)	Evaluates the expression and returns the day number of the week, where 1 = Monday.
DAYOFYEAR(<*expression*>)	Evaluates the expression and returns the day number of the year.
DAYS(<*expression*>)	Evaluates the expression and returns the number of days since January 1, 0001, plus 1.

Function Name	Description
HOUR(<*expression*>)	Evaluates the expression and returns the hour part of a time or timestamp.
JULIAN_DAY(<*expression*>)	Evaluates the expression and returns the number of days from January 1, 4713 B.C., to the date specified in the expression.
LAST_DAY(<*expression*>)	Returns the last day of the month indicated by the date in the expression.
MICROSECOND(<*expression*>)	Evaluates the timestamp or duration in the expression and returns the microsecond part.
MIDNIGHT_SECONDS(<*expression*>)	Evaluates the time or timestamp and returns the number of seconds between midnight and the time in the expression.
MINUTE(<*expression*>)	Evaluates the expression and returns the minute part of a time, timestamp, or time interval.
MONTH(<*expression*>)	Evaluates the expression and returns the month part of a date, timestamp, or date interval.
NEXTDAY(<*expression*>, <*dayname*>)	Evaluates the expression and returns as a timestamp the date of the first day specified in <*dayname*> (a string containing "MON", "TUE", etc.) following the date in the expression.
QUARTER(<*expression*>)	Evaluates the expression and returns the number of the quarter part of a year in which the date in the expression occurs.
ROUND_TIMESTAMP(<*expression*>, <*format string*>)	Evaluates the expression and rounds it to the nearest interval specified in the format string.
SECOND(<*expression*>)	Evaluates the expression and returns the seconds part of a time, timestamp, or time interval.
TIME(<*expression*>)	Evaluates the expression and returns the time part of a time or timestamp value.
TIMESTAMP(<*expression1*>, <*expression2*>)	Converts separate date (expression1) and time (expression2) values into a timestamp.
TRUNC_TIMESTAMP(<*expression*>, <*format string*>)	Evaluates the expression and truncates it to the nearest interval specified in the format string.
WEEK(<*expression*>)	Evaluates the expression and returns the week number of the date part of the value, where January 1 starts the first week.
WEEK_ISO(<*expression*>)	Evaluates the expression and returns the week number of the date part of the value, where the first week of the year is the first week containing a Thursday.
YEAR(<*expression*>)	Evaluates the expression and returns the year part of a date or timestamp.

Microsoft Office Access

Function Name	Description
CDate(<*expression*>)	Converts the expression to a date value.
Date	Obtains the current date value.
DateAdd(<*interval*>, <*number*>, <*expression*>)	Adds the specified number of the interval to the date or Date/Time expression.
DateDiff(<*interval*>, <*expression1*>, <*expression2*>, <*firstdayofweek*>, <*firstdayofyear*>)	Returns the specified number of intervals between the Date/Time in expression1 and the Date/Time in expression2. You can optionally specify a first day of the week other than Sunday and a first week of the year to start on January 1, the first week that has at least four days, or the first full week.
DatePart(<*interval*>, <*expression*>, <*firstdayofweek*>, <*firstdayofyear*>)	Extracts the part of the date or time from the expression as specified by the interval. You can optionally specify a first day of the week other than Sunday and a first week of the year to start on January 1, the first week that has at least four days, or the first full week.
DateSerial(<*year*>, <*month*>, <*day*>)	Returns the date value corresponding to the specified year, month, and day.
DateValue(<*expression*>)	Evaluates the expression and returns a Date/Time value.
Day(<*expression*>)	Evaluates the expression and returns the day part of a date.
Hour(<*expression*>)	Evaluates the expression and returns the hour part of a time.
IsDate(<*expression*>)	Evaluates the expression and returns True if the expression is a valid date value.
Minute(<*expression*>)	Evaluates the expression and returns the minute part of a time value.
Month(<*expression*>)	Evaluates the expression and returns the month part of a date value.
MonthName(<*expression*>, <*abbreviate*>)	Evaluates the expression (which must be an integer value from 1 to 12) and returns the equivalent month name. The name is abbreviated if the abbreviate argument is True.
Now	Obtains the current date and time value in the local time zone.
Second(<*expression*>)	Evaluates the expression and returns the seconds part of a time.
Time	Obtains the current time value in the local time zone.

Function Name	Description
TimeSerial(<*hour*>, <*minute*>, <*second*>)	Returns the time value corresponding to the specified hour, minute, and second.
TimeValue(<*expression*>)	Evaluates the expression and returns the time portion.
WeekDay(<*expression*>, <*firstdayofweek*>)	Evaluates the expression and returns an integer representing the day of the week. You can optionally specify a first day of the week other than Sunday.
WeekDayName(<*daynumber*>, <*abbreviate*>, <*firstdayofweek*>)	Returns the day of the week according to the specified day number. You can optionally ask to have the name abbreviated, and you can specify a first day of the week other than Sunday.
Year(<*expression*>)	Evaluates the expression and returns the year part of a date.

Microsoft SQL Server

Function Name	Description
CURRENT_TIMESTAMP	Obtains the current date and time in the local time zone.
DateAdd(<*interval*>, <*number*>, <*expression*>)	Adds the specified number of the interval to the date or datetime expression.
DateDiff(<*interval*>, <*expression1*>, <*expression2*>)	Returns the specified number of intervals between the datetime in expression1 and the datetime in expression2.
DateName(<interval>, <*expression*>)	Evaluates the expression and returns a string containing the name of the interval specified. If the interval is a month or a day of a week, the name is spelled out.
DatePart(<interval>, <*expression*>)	Extracts the part of the date or time from the expression as specified by the interval.
Day(<*expression*>)	Evaluates the expression and returns the day part of a date.
GetDate()	Obtains the current date.
GetUTCDate()	Obtains the current UTC (Coordinated Universal Time) date.
IsDate(<*expression*>)	Evaluates the expression and returns True if the expression is a valid date value.
Month(<*expression*>)	Evaluates the expression and returns the month part of a date value.
Year(<*expression*>)	Evaluates the expression and returns the year part of a date.

MySQL

Function Name	Description
ADDDATE(<*expression*>, <*days*>)	Adds the specified number of days to the date value in the expression.
ADDTIME(<*expression*>, <*time*>)	Adds the specified amount of time to the date or date-time expression value.
CURRENT_DATE	Obtains the current date value.
CURRENT_TIME	Obtains the current time value in the local time zone.
CURRENT_TIMESTAMP	Obtains the current date and time in the local time zone.
DATE(<*expression*>)	Extracts the date from a datetime expression.
DATEDIFF(<*expression1*>, <*expression2*>)	Subtracts the second datetime expression from the first datetime expression and returns the number of days between the two.
DATE_ADD(<*expression*>, INTERVAL <*interval*> <*quantity*>)	Adds the specified interval quantity to the date or date-time value in the expression.
DATE_SUB(<*expression*>, INTERVAL <*interval*> <*quantity*>)	Subtracts the specified interval quantity from the date or datetime value in the expression.
DAY(<*expression*>)	Evaluates the expression and returns the day part of a date as a number from 1 to 31.
DAYNAME(<*expression*>)	Evaluates the expression and returns the day name of the date or datetime value.
DAYOFMONTH(<*expression*>)	Evaluates the expression and returns the day part of a date as a number from 1 to 31.
DAYOFWEEK(<*expression*>)	Evaluates the expression and returns the day number within the week for the date or datetime value, where 1 = Sunday.
DAYOFYEAR(<*expression*>)	Evaluates the expression and returns the day number of the year, a value from 1 to 366.
EXTRACT(<*unit*> FROM <*expression*>)	Evaluates the expression and returns the unit portion (such as year or month) specified.
FROM_DAYS(<*number*>)	Returns the date that is the number of days since December 31, 1 B.C. Day 366 is January 1, 01.
HOUR(<*expression*>)	Evaluates the expression and returns the hour part of a time.
LAST_DAY(<*expression*>)	Returns the last day of the month indicated by the date in the expression.
LOCALTIME, LOCALTIMESTAMP	See the NOW function.

Function Name	Description
MAKEDATE(<*year*>, <*dayofyear*>)	Returns a date for the specified year and day of year (1–366).
MAKETIME(<*hour*>, <*minute*>, <*second*>)	Returns a time for the specified hour, minute, and second.
MICROSECOND(<*expression*>)	Evaluates the expression and returns the microsecond portion of a time or datetime value.
MINUTE(<*expression*>)	Evaluates the expression and returns the minute part of a time or datetime value.
MONTH(<*expression*>)	Evaluates the expression and returns the month part of a date value.
MONTHNAME(<*expression*>)	Evaluates the expression and returns the name of the month of a date or datetime value.
NOW()	Obtains the current date and time value in the local time zone.
QUARTER(<*expression*>)	Evaluates the expression and returns the number of the quarter part of a year in which the date in the expression occurs.
SECOND(<*expression*>)	Evaluates the expression and returns the seconds part of a time or datetime value.
STR_TO_DATE(<*expression*>, <*format*>)	Evaluates the expression according to the format specified and returns a date, datetime, or time value.
SUBDATE(<*expression*>, INTERVAL <*interval*> <*quantity*>)	See the DATE_SUB function.
SUBTIME(<*expression1*>, <*expression2*>)	Subtracts the time in expression2 from the datetime or time in expression1 and returns a time or datetime answer.
TIME(<*expression*>)	Evaluates the expression and returns the time part of a time or datetime value.
TIMEDIFF(<*expression1*>, <*expression2*>)	Subtracts the time or datetime value in expression2 from the time or datetime value in expression1 and returns the difference.
TIMESTAMP(<*expression*>)	Evaluates the expression and returns a datetime value.
TIMESTAMP(<*expression1*>, <*expression2*>)	Adds the time in expression2 to the date or datetime in expression1 and returns a datetime value.
TIMESTAMPADD(<*interval*>, <*number*>, <*expression*>)	Adds the specified number of the interval to the date or datetime expression.
TIMESTAMPDIFF(<*interval*>, <*expression1*>, <*expression2*>)	Returns the specified number of intervals between the date or datetime in expression1 and the date or datetime in expression2.

Function Name	Description
TIME_TO_SEC(<*expression*>)	Evaluates the time in the expression and returns the number of seconds.
TO_DAYS(<*expression*>)	Evaluates the date in the expression and returns the number of days since year 0.
UTC_DATE	Obtains the current UTC (Coordinated Universal Time) date.
UTC_TIME	Obtains the current UTC (Coordinated Universal Time) time.
UTC_TIMESTAMP	Obtains the current UTC (Coordinated Universal Time) date and time.
WEEK(<*expression*>, <*mode*>)	Evaluates the expression and returns the week number of the date part of the value using the mode specified.
WEEKDAY(<*expression*>)	Evaluates the expression and returns an integer representing the day of the week, where 0 is Monday.
WEEKOFYEAR(<*expression*>)	Evaluates the date expression and returns the week number (1–53), assuming the first week has more than three days.
YEAR(<*expression*>)	Evaluates the expression and returns the year part of a date.

Oracle

Function Name	Description
CURRENT_DATE	Obtains the current date value.
CURRENT_TIMESTAMP	Obtains the current date, time, and timestamp in the local time zone.
DBTIMEZONE	Obtains the time zone of the database.
EXTRACT(<*interval*> FROM <*expression*>)	Evaluates the expression and returns the requested interval (year, month, day, etc.).
LOCALTIMESTAMP	Obtains the current date and time in the local time zone.
MONTHS_BETWEEN(<*expression1*>, <*expression2*>)	Calculates the months and fractions of months between expression2 and expression1.
NEW_TIME(<*expression*>, <*timezone1*>, <*timezone2*>)	Evaluates the date expression as though it is the first time zone and returns the date in the second time zone.
NEXT_DAY(<*expression*>, <*dayname*>)	Evaluates the expression and returns the date of the first day specified in <dayname> (a string containing "MONDAY", "TUESDAY", etc.) following the date in the expression.

Function Name	Description
NUMTODSINTERVAL(<*number*>, <*unit*>)	Converts the number to an interval in the unit specified (DAY, HOUR, MINUTE, SECOND).
NUMTOYMINTERVAL(<*number*>, <*unit*>)	Converts the number to an interval in the unit specified (YEAR, MONTH).
ROUND(<*expression*>, <*interval*>)	Rounds a date value to the interval specified.
SESSIONTIMEZONE	Obtains the time zone of the current session.
SYSDATE	Obtains the current date and time on the database server.
SYSTIMESTAMP	Obtains the current date, time, and time zone on the database server.
TO_DATE(<*expression*>, <*format*>)	Converts the string expression to a date data type using the format specified.
TO_DSINTERVAL(<*expression*>)	Converts the string expression to a days-to-seconds interval.
TO_TIMESTAMP(<*expression*>, <*format*>)	Converts the string expression to a timestamp data type using the format specified.
TO_TIMESTAMP_TZ(<*expression*>, <*format*>)	Converts the string expression to a timestamp with time zone data type using the format specified.
TO_YMINTERVAL	Converts the string expression to a years-to-months interval.
TRUNC(<*expression*>, <*interval*>)	Truncates a date value to the interval specified.

D

Suggested Reading

These are the books we recommend you read if you want to learn more about database design or expand your knowledge of SQL. Keep in mind that some of these books will be challenging because they are more technical in nature. Also, some authors assume that you have a fairly significant background in computers, databases, and programming.

Database Books

Connolly, Thomas, and Carolyn Begg. *Database Systems: A Practical Approach to Design, Implementation, and Management* (4th ed.). Essex, England: Addison-Wesley, 2004.

Date, C. J. *An Introduction to Database Systems* (8th ed.). Boston, MA: Addison-Wesley, 2003.

———. *Database in Depth: Relational Theory for Practitioners*. Sebastopol, CA: O'Reilly Media, 2005.

Hernandez, Michael J. *Database Design for Mere Mortals* (2nd ed.). Boston, MA: Addison-Wesley, 2003.

Books on SQL

Bowman, Judith S., Sandra L. Emerson, and Marcy Darnovsky. *The Practical SQL Handbook* (4th ed.). Boston, MA: Addison-Wesley, 2001.

Celko, Joe. *Joe Celko's SQL for Smarties: Advanced SQL Programming* (3rd ed.). San Francisco, CA: Morgan Kaufmann Publishers, 2005.

Date, C. J., and Hugh Darwen. *A Guide to the SQL Standard* (4th ed.). Reading, MA: Addison-Wesley, 1997.

Gruber, Martin. *SQL Instant Reference* (2nd ed.). Alameda, CA: Sybex Inc., 2000.

Melton, Jim, and Alan R. Simon. *Understanding the New SQL: A Complete Guide*. San Francisco, CA: Morgan Kaufmann Publishers, 2006.

This page intentionally left blank

Index

A
abbreviations
  in field names, 22
  in table names, 31
Access. See under Microsoft
acronyms
  in field names, 22
  in table names, 31
aggregate functions, 74, 375–377, 416–428. *See also* AVG; COUNT; COUNT(*); MAX; MIN; SUM
  column name in, 419
  in filters, 428–430
  grouping of, 442–444 (*See also* GROUP BY)
  imbedding of, 428
  multiple, 427–428
  Nulls and, 417, 420
  overview, 416–418
  sample statements, 431–438
  syntax, 416, 416*f,* 595*f*
  uses of, 416
  WHERE clause in, 419, 430
aliases
  for column names, 131
  for tables, 252–254, 252*f*
ALL
  in subqueries, 386–389, 387*f,* 393–394, 406–407
    syntax, 487*f,* 598*f*
  in UNIONs, 228–230, 341, 343, 345, 348
Allaire, *Cold Fusion,* 6
American National Standards Institute (ANSI), 57. *See also* SQL Standard
American Standard Code for Information interchange (ASCII), 107
analytic databases, defined, 4
AND, 179, 180*f*
  null values and, 195–196
  with OR, 183–184
    sample statements, 204–205
Ansa Software, *Paradox,* 5
ANSI (American National Standards Institute), 57
ANSI/ISO standard. *See* SQL Standard
ANSI NCITS-H2, 62
"ANSI X3.168-1989 Database Language Embedded SQL," 58
"ANSI X3.135-1986 Database Language SQL" (SQL/86), 57–58
ANY, 386–389, 387*f,* 400, 598*f*
approximate numeric data type, 108–109
archiving of data, 547–549, 554–555
  deleting data after archiving, 574–575, 579
artificial primary keys, 41–42, 41*f*
AS, 130–131, 252–253, 252*f*
ASC, 91, 96, 98–99, 100–101
ASCII (American Standard Code for Information interchange), 107, 157–159
Ashton Tate, *dBase,* 5
asterisk shortcut, 83–84, 83*f*
ASYMMETRIC BETWEEN, 165
attribute, defined, 6, 215. *See also* fields
AUTO_INCREMENT, 543
AutoNumber *(Access),* 543
AVG, 416*f,* 423–424, 432, 435, 595*f*. *See also* aggregate functions

B
base tables, defined, 9
Begg, Carolyn, 50
BETWEEN, 154, 164–167, 164*f,* 191–193
  ASYMMETRIC, 165
  SYMMETRIC, 165
BIGINT data type, 108
BINARY data type, 108
BINARY LARGE OBJECT data type, 108

BIT data type, 108, 109
BIT_LENGTH, 595*f*
BIT VARYING data type, 108
BLOB data type, 108
Boolean data type, 109

C
calculated columns, naming of, 129–131, 130*f*
calculated data. *See also* expressions
  in fields
    disadvantages of, 24, 25
    resolving of, 33
    uses of, 514–516
  in SELECT statements, 74
Call-Level Interface (CLI) specification, 61–62
Cartesian product, 248, 297
cascade deletion rule, 45, 46*f*
case sensitivity
  in collation of character string literals, 157–160
  in comparison of character string literals, 171
CAST, 110–112, 110*f*
  for datetime literal compatibility, 117, 119–120, 141, 142–143, 144–145
  for mathematical expression compatibility, 123–124
  syntax, 110*f,* 594*f*
Chamberlin, Donald, 54
changing data. *See* DELETE; INSERT; UPDATE
CHARACTER (CHAR) data type, 107
CHARACTER (CHAR) LARGE OBJECT data type, 107
CHARACTER (CHAR) VARYING data type, 107
character string functions, syntax, 594*f*
character string literals
  collating sequence of, 157–160
  concatenating, 118–119, 118*f*
  less than and greater than comparisons, 162
  LIKE and, 169–173, 170*f*
  MAX and, 424
  range condition predicates, 166
  specifying, 112–114, 113*f*
  syntax, 113*f,* 492*f*
CHAR_LENGTH, 595*f*
CLI (Call-Level Interface) specification, 61–62
client/server computing, introduction of, 5, 65
CLOB data type, 107
Codd, Edgar F., 4–5, 54
Cold Fusion, 6
collating sequence
  of character string literals, 157–160
  ORDER BY clause and, 88–89
columns
  added, SELECT statement and, 84
  aliases for, 131
  calculated, naming of, 129–131, 130*f*
  column references, 245–246, 246*f,* 249, 593*f*
  deleted, SELECT statement and, 84
  ordering of, 82–83
  as term, 72
COMMIT, 506
comparison predicates, 156–164
  comparison operators, 154
  equality and inequality, 160–161
  exclusive, 161
  inclusive, 161
  less than and greater than, 162–164
  NOT operator and, 185–186
  string values collating sequence, 157–160
  syntax, 157*f,* 597*f*
  types of, 156–157, 157*f*
composite primary keys (CPK)
  defined, 8, 39
  use of, 39
Computer Associates International, Inc., 56
concatenation expressions, 117–120, 118*f*
  length limits for returns, 119
  in SELECT clause, 128–129
  storing as data, 24
  syntax, 118*f*
concatenation operators, 117–118, 118*f*
Connolly, Thomas, 50
copying of data, 547–549
correlation names (aliases)
  for column names, 131
  for tables, 252–254, 252*f*
CORRESPONDING clause, 343
COUNT, 375–376, 420–421. *See also* aggregate functions

DISTINCT and, 420, 469
HAVING COUNT trap, 481–485
Null values and, 420
sample statements, 427–428, 431, 435
syntax, 376*f,* 416*f,* 595*f*
COUNT(*), 375–376, 418–420
DISTINCT and, 421
Null values and, 417
sample statements, 395, 397, 399, 437, 438
syntax, 376*f,* 416*f,* 595*f*
WHERE clause of, 419
CPK. *See* composite primary keys

D
data. *See also* data types
defined, 75
dynamic, defined, 4
static, defined, 4
vs. information, 75–77, 76*f*
Database 2 (DB2). *See under* IBM
database design. *See also* database structure
books on, 16–17
importance of understanding, 19–20
methodology, good
analysis of database with, 50
value of, 16
vs. database theory, 16
vs. implementation, 16
Database Design for Mere Mortals (Hernandez), 5, 16, 42, 50
database implementation, *vs.* database design, 16
database models
relational, history of, 4–6
types of, 4
databases
examples
downloadable version of, 93
schema of, 601–605
types of, 3–4
Web-centric, 6, 65
database structure
fine-tuning
fields, 21–29
tables, 30–42
thorough design process analysis, 50
sound, importance of, 20
Database Systems: A Practical Approach (Connolly and Begg), 50
database theory
importance of understanding, 16
vs. database design, 16
data types, 107–109
changing types, 110–112, 110*f*
extended, 109
Date, C. J., 16, 58
DATE data type, 109
date expressions, 124–126, 125*f*
leap year adjustments, 144–145
sample expressions, 141, 142–143, 144–145
in SELECT clause, 132–133
date functions
in specific software, 607–614
syntax, 594*f*
DATE keyword, 116, 117
date literals
BETWEEN and, 165–166, 167
specifying, 115–116
syntax, 115*f,* 592*f*
datetime arithmetic expressions, 124–127
DATETIME data type. *See also entries under* date; time; timestamp
concatenation of, 119–120
overview of, 109
datetime functions
in specific software, 607–614
syntax, 594*f*
datetime literals
specifying, 115–117, 115*f*
syntax, 115*f,* 592*f*
dBase, 5
DB2 (Database 2). *See under* IBM
DEC data type, 108
DECIMAL data type, 108
DEFAULT, 540
degree of participation
defined, 48
enforcement of, 49–50
setting, 48–50, 49*f*
DELETE
of all rows, 569–570
of data after archiving, 574–575, 579
overview, 567–568

DELETE (*cont.*)
recovering lost data, 506–507, 570
sample statements, 576–583
of some rows, 571–575
syntax, 568–569, 568*f,* 599*f*
transactions and, 506–507, 570
transforming SELECT statement into, 572, 572*f*
uses of, 568, 575–576
verifying accuracy of target materials, 571–572
WHERE clause in, 568, 568*f,* 571–575
sample statements, 576–583
subqueries, 573–575
deletion rule
defined, 44
establishing, 44–45, 46*f*
types of, 45, 46*f*
delimited identifiers, 22, 32
derived tables, in INNER JOIN, 254–256, 255*f*
DESC keyword, 90–91, 100
diagrams. *See* syntax diagrams
difference, 222–227. *See also* EXCEPT
commercial systems' support, 233
limitations of, 227–228
by OUTER JOIN with Null test, 295, 300, 318
between result sets, 224–227
set diagrams of, 225–227, 225*f,* 227*f*
in set theory, 222–224
uses of, 215, 227–228
DISTINCT
AVG and, 424
COUNT/COUNT(*) and, 376, 420, 469
in JOINs, 233, 247*f,* 252*f,* 255*f,* 296*f,* 302*f,* 305*f,* 358, 384
MAX and, 425–426
MIN and, 426–427
in SELECT statements, 84–86, 85*f,* 97, 102
in UNIONs, 348
"does not apply," *vs.* "is not applicable," 137–138
DOUBLE PRECISION data type, 108–109
duplicate fields
disadvantages of, 24–25, 36
identification of, 33
resolving of, 33–38
duplicate rows, eliminating, 84–86, 85*f*
dynamic data, defined, 4

E

EBCDIC (Extended Binary Coded Decimal Interchange Code), 158–159, 197
embedded SQL, specification for, 58–59
enforcement, of participation degree, 49–50
equality comparison predicates, 160–161
escape characters, 172
ESCAPE option, of LIKE predicate, 169*f,* 172–173
Euler, Leonard, 219
Euler diagrams, 219. *See also* set diagrams
events, defined, 7, 7*f,* 32–33
exact number data type, 108
EXCEPT, 222–227
alternatives to, 295
syntax, 236–239, 238*f*
execution, of query
defined, 93
methods, 93
EXISTS, 389–392, 393–394, 402–403, 598*f*
expressions. *See also* value expressions
column, subqueries as, 372–377
aggregate functions for, 375–377, 376*f*
GROUP BY in, 452–453, 461–463
sample statements, 395–499
syntax, 372–375
uses, 392
data types
changing of, 110–112, 110*f* (*See also* CAST)
overview of, 107–109
defined, 106
literal values, specifying, 112–117
character string literals, 112–114, 113*f*
datetime literals, 115–117, 115*f*
numeric literals, 114*f*
naming of, 129–131, 130*f*
overview of, 106
in SELECT clause, 128–135
concatenation expressions, 128–129
date expressions, 132–133
mathematical expressions, 131–132
sample statements, 139–147
value expressions, 135, 135*f*

types of, 117–127
concatenation, 117–120, 118*f*
length limits for returns, 119
in SELECT clause, 128–129
storing as data, 24
syntax, 118*f*
date and time, 124–127
leap year adjustments, 144–145
sample expressions, 141, 142–143, 144–145
in SELECT clause, 132–133
mathematical, 121–124, 121*f*
nulls in, 138–139, 139*f*
sample statements, 140, 143, 145, 147
in SELECT clause, 131–132
uses of, 128
Extended Binary Coded Decimal Interchange Code (EBCDIC), 107, 158–159
extended data types, 109
extensions to SQL standard, 60–61, 73
EXTRACT, 595*f*

F
false, value of, 507–508
Federal Information Processing Standard (FIPS), 61
fields
duplicate
disadvantages of, 24–25, 36
identification of, 33
resolving of, 33–38
fine-tuning of, 21–29
multipart
identification of, 24, 25, 25*f,* 26, 27*f*
resolving of, 25–27, 33
multivalued
identification of, 24, 27–28, 27*f*
resolving of, 27–29, 28*f,* 33
naming of, 21–22
overview of, 7–8
structure, fine-tuning of, 23–25, 23*f*
as term, 72
filters. *See also* HAVING clause; WHERE clause
aggregate functions in, 428–430
subqueries as, 377–392
predicate keywords, 380–392
sample statements, 400–409
syntax, 378–380, 378*f*
uses, 393–394
WHERE *vs.* HAVING, uses of, 478–485
FIPS (Federal Information Processing Standard), 61
FIPS PUB 127, 61
FK (foreign keys), 9, 9*f*
FLOAT data type, 108–109
foreign keys (FK), 9, 9*f*
FROM clause
in FULL OUTER JOIN, 314*f*
in INNER JOIN, 247, 247*f,* 249–250
correlation names for tables, 252–254, 252*f*
embedded JOIN in, 256–261, 257*f,* 260*f*
sample statements, 270–272, 276–277, 280–282, 284–288
embedded select statement in, 254–256, 255*f*
sample statements, 278–282, 284–288
in OUTER JOIN, 296–300, 296*f*
embedded JOIN in, 304–314, 305*f,* 306*f,* 308*f*
sample statements, 320–324, 326–331, 333–334
embedded select statements in, 301–304, 302*f*
sample statements, 320–322, 326–327, 329–331
in SELECT, 74, 74*f*
in UNION JOIN, 318*f*
FULL OUTER JOIN, 314–317, 314*f*
alternatives to, 316, 334
on non-key values, 317
function, defined, 73, 92

G
greater than comparison predicates, 162–164
GROUP BY, 444–458
aggregate functions in, 452, 455, 457–458
mixing columns and expressions, 450–452
nonaggregate column references, 450, 452, 455–456
sample statements, 459–470
as SELECT DISTINCT alternative, 453–454

GROUP BY (*cont.*)
in SELECT statement, 74*f,* 75
in subquery, 452–453, 461–463
syntax, 445–450, 445*f*
uses for, 458
without aggregate functions, 453–454

H
HAVING clause, 74*f,* 75
HAVING COUNT trap, 481–485
restrictions on, 476
sample statements, 487–496
syntax, 476, 476*f*
uses of, 474–478, 486–487
vs. WHERE, uses of, 478–485
HAVING COUNT trap, 481–485
Hernandez, Michael J., 5, 16, 42, 50

I
IBM
DB2 (Database 2)
concatenation in, 118
data entry in, 538
date and time functions in, 607–608
history of, 5, 56, 57
not equal to operator, 161
SAA (Systems Application Architecture), 61
SQL/Data System (SQL/DS), 56
SQL development, 54–55
string values collating sequence, 158–159
System R, 5, 54–55
identifiers
delimited, 22, 32
regular, 22, 32
inequality comparison predicates, 160–161
information
defined, 75
vs. data, 75–77, 76*f*
Informix-SE, 65, 118
INGRES, 5, 56, 57, 118
IN keyword, 380–386, 381*f,* 393–394, 404
INNER JOIN, 244–262
ON clause, 247–250, 247*f*
column references, 245–246, 246*f,* 249
correlation names for tables, assigning, 252–254, 252*f*
embedded JOINS in, 256–261, 257*f,* 260*f*
sample statements, 270–272, 276–277, 280–282, 284–288
embedded SELECT statements in, 254–256, 255*f*
sample statements, 278–282, 284–288
JOIN-eligible data types, 244–245
knowledge of tables and, 261–262
overview of, 244
sample statements, 263–288
syntax, 246–262, 247*f,* 257*f*
of three or more tables, 256–261, 257*f,* 260*f,* 270–277, 347*f*
of two tables, 247–251, 247*f,* 257*f,* 264–269, 339–340, 340*f*
uses for, 262–263
USING clause, 247, 247*f,* 250–251
IN predicate, 154, 167–169, 168*f,* 598*f*
INSERT
column references in, 540
individual values and rows, 539–544, 539*f*
overview, 537–539
primary keys, automatic generation of, 542–544
sample statements, 552–563
with SELECT expression, 544–550, 544*f,* 545*f,* 599*f*
sample expressions, 554–555, 556–559, 561–562
sets of data, 544–550, 544*f,* 545*f*
transactions and, 506
uses for, 550–551
with VALUES keyword, 539–544
sample statements, 552–553, 556, 559–560
syntax, 539*f,* 599*f*
INTEGER (INT) data type, 108, 109
International Organization for Standardization (ISO), 58. *See also* SQL Standard
INTERSECT, syntax, 234–236, 236*f*
intersection, 216–222. *See also* INTERSECT
commercial systems' support, 233
limitations of, 221
set diagrams of, 218–221, 219*f,* 220*f,* 235*f*
of set results, 217–221
in set theory, 216–217
uses of, 215, 221–222
INTERVAL data type, 109, 125

interval literals, 116, 593*f*
An Introduction to Database Systems (Date), 16
"is not applicable," *vs.* "does not apply," 137–138
IS NULL predicate, 154, 173–175, 174*f,* 197, 205, 295, 300, 318
  syntax, 174*f,* 598*f*
"ISO 9075-1987 Database Language SQL," 58
"ISO 9075:1989 Database Language SQL with Integrity Enhancements" (SQL/89), 58
"ISO/IEC 9075:1992 Database Language SQL," 59–60
ISO (International Organization for Standardization), 58. *See also* SQL standard

J
JOIN. *See also* INNER JOIN; OUTER JOIN
  alternatives to, 250
  commercial systems' support, 250
  default mode, 247
  NATURAL, 251, 301
  overview of, 243–244
  UNION, 317, 318*f*
  uses of, 221–222

K
keys
  composite primary (CPK)
    defined, 8, 39
    use of, 39
  foreign (FK), 9, 9*f*
  overview of, 8–9
  primary (PK)
    artificial, 41–42, 41*f*
    automatic generation of, 542–544
    composite, 8, 39
    criteria for, 39–41, 40*f,* 41*f*
    defined, 6, 8, 9*f,* 39
    functions of, 8, 33
    simple, 39

L
largest value. *See* MAX
LEFT OUTER JOIN
  defined, 295–296
  sample statements, 319–334
  syntax, 296–314, 296*f*
less than comparison predicates, 162–164
LIKE, 154, 169–173, 169*f*
  ESCAPE option, 169*f,* 172–173
  sample pattern strings, 170*t*
  sample statements, 206
  syntax, 169*f,* 598*f*
  wildcard characters, 169–170
linking tables
  advantages of, 14–15
  defined, 13
  defining of, 13–14
  in resolving duplicate fields, 37–38, 37*f,* 38*f*
  in resolving multivalued fields, 28, 29*f*
literals. *See also* character string literals; date literals; datetime literals; numeric literals; time literals; timestamp literals
  interval, 116, 593*f*
  keywords for, 116, 117
  specifying values for
    character string literals, 112–114, 113*f*
    datetime literals, 115–117, 115*f*
    numeric literals, 114*f*
  types
    changing, 110–112, 110*f* (*See also* CAST)
    overview, 107–109
lost data, recovering, 570. *See also* transactions

M
mandatory participation, 46–47, 48*f*
many-to-many relationship, 13–15, 13*f,* 14*f,* 43–44, 44*f*
mathematical expressions, 121–124, 121*f*
  nulls in, 138–139, 139*f*
  sample statements, 140, 143, 145, 147
  in SELECT clause, 131–132
MAX, 376–377, 424–426. *See also* aggregate functions
  in multiple aggregate functions, 427
  sample statements, 396, 398, 434
  syntax, 376*f,* 416*f,* 595*f*
mean values. *See* AVG
members, of set. *See also* IN predicate
  characteristics of, 216–217
  defined, 215

MEMO data type, 107
Microrim, 5
Microsoft
  Access, 29, 65, 92
    AutoNumber in, 543
    concatenation in, 118
    COUNT DISTINCT and, 470
    date and time functions in, 609–610
    date entry in, 537–538
    DELETE in, 570, 570*f*
    GROUP BY in, 456
    JOINs in, 320, 334
    LIKE predicate in, 170
    subqueries, aggregate functions in, 521
    transactions in, 506
    UPDATE
      JOIN clause in, 511–512, 521
  subquery UPDATE expressions, 514
  ODBC (Open Database Connectivity) specification, 62
  SQL Server
    concatenation in, 118
    date and time functions in, 610
    FULL OUTER JOINs in, 316
    history of, 5–6, 65
    Identity data type, 543
    not equal to operator, 161
    OUTER JOIN in, 299
    TOP keyword in, 92
    UPDATE, JOIN clause in, 511–512
  Visual Studio, 6
MIN, 416*f,* 426–427, 433, 595*f*. *See also* aggregate functions
MONEY/CURRENCY data type, 109
multipart fields
  identification of, 24, 25, 25*f,* 26, 27*f*
  resolving of, 25–27, 33
multiple column retrieval, 81–83, 81*f*
multivalued fields
  identification of, 24, 27–28, 27*f*
  resolving of, 27–29, 28*f,* 33
MySQL
  AUTO_INCREMENT in, 543
  concatenation in, 118
  data entry in, 538
  date and time functions in, 611–613
  datetime specification in, 116, 117
  embedded SELECT statements in, 255
  history of, 6
  JOINs in, 334

N
names. *See also* aliases
  for columns, 130–131
  of fields, 21–22
  of tables, 30–32
NATIONAL CHARACTER (CHAR) data type, 107–108
NATIONAL CHARACTER (CHAR) LARGE OBJECT data type, 108
NATIONAL CHARACTER (CHAR) VARYING data type, 108
National Committee for Information Technology Standards (NCITS), 62
National Institute of Standards and Technology (NIST), 61
NATURAL JOIN, 251, 301
NCHAR data type, 107–108
NCHAR VARYING data type, 108
NCITS-H2, 62
NCITS (National Committee for Information Technology Standards), 62
NCLOB data type, 108
NEXTVAL, 543
NIST (National Institute of Standards and Technology), 61
not equal to operator, 161
NOT operator
  in comparison conditions, 185–186
  double, 186–187
  as first keyword, 184–187, 185*f*
  within predicate, 175–178, 176*f*
  sample statements, 186–187
NTEXT data type, 108
Nulls, 135–139. *See also* IS NULL predicate
  COUNT and, 420
  COUNT(*) and, 417
  defined, 136–137
  in mathematical expressions, 138–139, 139*f*
  and multiple condition searches, 193–197, 194*f,* 195*f,* 196*f*
  uses of, 137–138
NUMERIC data type, 108

numeric functions, syntax, 595*f*
numeric literals
  BETWEEN and, 164–165
  specifying, 114*f*
  syntax, 114*f,* 592*f*

O
objects, defined, 7, 32
ODBC (Open Database Connectivity) specification, 62
Office Access 2007, 29
ON clause, 247–250, 247*f,* 296*f,* 297–301
one-to-many relationship, 12, 12*f,* 43, 43*f*
one-to-one relationship, 11–12, 12*f,* 42–43, 43*f*
Open Database Connectivity (ODBC) specification, 62
operational databases, defined, 4
optional participation, 46–47, 48*f*
Oracle
  concatenation in, 118
  data entry in, 538
  date and time functions in, 613–614
  GROUP BY in, 456
  history of, 5, 55–56
  NEXTVAL in, 543
  OUTER JOIN in, 299
ORDER BY
  ASC keyword, 91, 96, 98–99, 100–101
  column name, specifying, 90–91, 351–352, 351*f*
  column number, specifying, 351–352, 351*f*
  DESC keyword, 90–91, 100
  order of precedence in, 88–89
  in SELECT statements, 87–92, 87*f*
    sample statements, 96, 98–99, 100–101, 146, 200–202
  in UNIONs, 351–352, 351*f*
ordering
  of columns, 82–83
  of rows, 87–92, 87*f*
order of precedence
  in mathematical expressions, 121–123
  in multiple search conditions, 182
    default order, 187–188
    and efficiency of search, 190–191
    prioritizing of conditions, 188–190
  in ORDER BY clause, 88–89
OR operator, 180–182, 181*f*
  with AND, 183–184
  null values and, 196, 196*t*
  sample statements, 203
orphaned records
  avoiding, 9, 44–45
  defined, 44
OUTER JOIN
  ON clause, 296*f,* 297–301
  commercial systems' support, 299
  embedded JOINS in, 304–314, 305*f,* 306*f,* 308*f*
    sample statements, 320–324, 326–331, 333–334
  embedded SELECT statements in, 301–304, 302*f*
    sample statements, 320–322, 326–327, 329–331
  FULL, 314–317, 314*f*
    alternatives to, 316, 334
    on non-key values, 317
  LEFT
    defined, 295–296
    sample statements, 319–334
    syntax, 296–314, 296*f*
  overview, 293–295
  RIGHT
    defined, 295–296
    syntax, 296–314, 296*f*
  sample statements, 319–334
  syntax, 296–314, 296*f*
  of three of more tables, 304–314, 306*f,* 308*f*
  of two tables, 296–301, 296*f,* 305*f*
  uses of, 228, 318–319
  USING clause, 296*f,* 297, 300–301
overlapping ranges, checking for, 191–193, 192*f*

P
Paradox, 5
parent-child relationship, 311
parentheses
  for combining search conditions, 184
  in embedded JOINS within JOINS, 257–258, 257*f,* 304–305, 306*f,* 308*f*
  in embedded SELECT statements within JOINS, 254–255, 255*f*
  in mathematical expressions, 122–123

parentheses (*cont.*)
in prioritizing of conditions, 189–190
for subqueries, 374
participation degree
defined, 48
enforcement of, 49–50
setting, 48–50, 49*f*
participation type, setting, 46–47, 47*f,* 48*f*
pattern-matching condition. *See* LIKE
PK. *See* primary keys
POSITION, 595*f*
predicates. *See also* search conditions
quantified, 386–389, 387*f*
in WHERE clause (*See also* BETWEEN; comparison predicates; IN predicate; IS NULL predicate; LIKE)
efficiency of, 190–191
expressing in different ways, 197–198
INTERSECT, 234–236, 236*f*
multiple conditions, 178–197
NOT operator, 175–178, 176*f,* 184–187, 185*f*
nulls, evaluation of, 193–197, 194*f,* 195*f,* 196*f*
AND operator, 179, 180*f,* 183–184, 195–196, 204–205
order of precedence. *See* order of precedence
OR operator, 180–182, 181*f,* 184–185, 196, 196*t,* 203
AND and OR together, 183–184
overview, 152–154
sample statements, 198–206
primary keys (PK)
artificial, 41–42, 41*f*
automatic generation of, 542–544
composite, 8, 39
criteria for, 39–41, 40*f,* 41*f*
defined, 6, 8, 9*f,* 39
functions of, 8, 33
simple, 39
primary tables, defined, 11

Q

quantified predicates, 386–389, 387*f*
QUEL (Query Language), 56, 57
query. *See also* SELECT query
defined, 10, 73, 92–93
execution methods, 93
saved, defined, 10
Query Language (QUEL), 56, 57
query optimizers, 190–191, 256, 308–309, 320, 354
quotes
for character string literals, 112–113, 113*f*
single, embedded, 186

R

range, finding data within. *See* BETWEEN
R:BASE, 5
RDBMSs. *See* relational database management systems
REAL data type, 108–109
records
orphaned
avoiding, 9, 44–45
defined, 44
overview of, 8
as term, 72
recovering lost data, 570. *See also* transactions
regular identifiers, 22, 32
relation, 6. *See also* tables
relational database management systems (RDBMSs)
collating sequences of, 88–89
history of, 55–56, 64–65
products, 5–6
relational databases
anatomy of, 6–15
history of model, 4–6
importance of understanding, 15–16
relational database software. *See* relational database management systems
"A Relational Model of Data for Large Shared Databanks" (Codd), 5
Relational Software, Inc., 55–56
Relational Technology, Inc., 56
relationships between tables. *See also* linking tables
degree of participation
defined, 48
enforcement of, 49–50
setting, 48–50, 49*f*
deletion rule
defined, 44

establishing, 44–45, 46*f*
types of, 45, 46*f*
relationship characteristics, establishing, 44–50
type of participation, setting, 46–47, 47*f,* 48*f*
unresolved, defined, 13, 13*f*
Request/Translation/Clean UP/SQL technique, 77–81
restrict deletion rule, 45, 46*f*
result set, defined, 77
RIGHT OUTER JOIN
defined, 295–296
syntax, 296–314, 296*f*
ROLLBACK, 506
ROUND, 521, 522
rows
counting of. *See* COUNT(*)
ordering of, 87–92, 87*f*
as term, 72
row subqueries, 370–371
row value constructors, 370–371

S

SAA (Systems Application Architecture), 61
sample database schema, 601–605
saved query, defined, 10
saving, of SELECT statements, 92–93
scalar subqueries, 370, 372, 373–375
search conditions. *See also* HAVING clause; ON clause; predicates; WHERE clause
combining with parentheses, 184
expressing in different ways, 197–198
order of precedence, 182
default order, 187–188
and efficiency of search, 190–191
prioritizing of conditions, 188–190
syntax diagram, 178*f,* 185*f,* 597*f*
in WHERE clause, 152–153
secondary tables, defined, 11
SELECT clause
expressions in, 128–135
concatenation expressions, 128–129
date expressions, 132–133
mathematical expressions, 131–132
sample statements, 139–147
value expressions, 135, 135*f*
overview of, 74, 74*f*
SELECT expression
defined, 342
INSERT using, 544–550, 544*f,* 545*f*
as part of SELECT operation, 72
syntax, 591*f*
in value expression, 373, 373*f*
SELECT operation
overview, 72
parts of, 72 (*See also* SELECT expression; SELECT query; SELECT statement)
SELECT query, 87–92, 87*f*
defined, 87
as part of SELECT operation, 72
sample statements, 96, 98–99, 100–101, 146, 200–202
sorting rows with, 87–92, 87*f*
syntax, 87–88, 87*f,* 591*f*
SELECT statement
added or deleted columns and, 84
all columns, retrieval of, 83–84, 83*f*
clauses of, 73–75, 74*f* (*See also* FROM clause; GROUP BY; HAVING; SELECT clause; WHERE clause)
defining, 73
duplicate rows, eliminating, 84–86, 85*f*
embedded
in INNER JOIN, 254–256, 255*f*
sample statements, 278–282, 284–288
in OUTER JOIN, 301–304, 302*f*
sample statements, 320–322, 326–327, 329–331
keywords, 73
multiple columns, retrieval of, 81–83, 81*f*
ordering of columns, 82–83
ordering of rows, 87–92, 87*f*
overview of, 73
as part of SELECT operation, 72
sample statements, 93–102
saving, 92–93
subqueries in, 372–377
aggregate functions for, 375–377, 376*f*
GROUP BY in, 452–453, 461–463
sample statements, 395–499
syntax, 372–375
uses, 392
syntax, 74*f,* 77–78, 78*f,* 81*f,* 135*f,* 372*f*
transforming into DELETE statement, 572, 572*f*

SELECT statement (*cont.*)
transforming into UPDATE statement, 505, 506*f*
translating requests into SQL, 77–81
SEQUEL (Structured English Query Language), 54
SEQUEL-XRM, 54
SERIAL/ROWID data type, 109
set(s)
defined, 214–215
members of (*See also* IN predicate)
characteristics of, 216–217
defined, 215
operations on, 215 (*See also* difference; intersection; union)
set diagrams
difference, 225–227, 225*f,* 227*f*
intersection, 218–221, 219*f,* 220*f,* 235*f*
union, 231–232, 232*f*
set identifier columns, 348, 357, 391
set theory
difference in, 222–224
intersection in, 216–217
union in, 228–230
simple primary keys, 39
smallest value. *See* MIN
SMALLINT data type, 108
SOME, 386–389, 387*f,* 393–394, 598*f*
spaces, in expression names, 130
Specifying Queries as Relational Expressions (SQUARE), 55
SQL
early vendor implementations, 55–56
future of, 65
history of, 53–65
importance of learning, 65–66
pronunciation of, 54
SQL3, 62, 62*t*–64*t*
SQL/86, 57–58, 59
SQL/89, 58, 59
SQL/92, 59–60
SQL:1999, 65
SQL:2003, 65
SQL:2007, 65
SQL Access group, 61
SQL/Data System (SQL/DS), 56
SQL/DS (SQL/Data System), 56
SQL Server. See under Microsoft
SQL Standard
alternative standards, 61–62
database models and, 4
data types in, 107–109
datetime expressions and, 127
development of, 56–58
evolution of, 58–64
extensions to, 60–61, 73
identifiers, types of, 22, 32
ORDER BY clause in, 88
parts of, 62, 62*t*–64*t*
SQUARE (Specifying Queries as Relational Expressions), 55
START TRANSACTION, 506
static data, defined, 4
Stonebraker, Michael, 56
stored procedure, defined, 73, 92
Structured English Query Language (SEQUEL), 54
subqueries
for automatic primary keys generation, 543–544
as column expressions, 372–377
aggregate functions for, 375–377, 376*f*
GROUP BY in, 452–453, 461–463
sample statements, 395–499
syntax, 372–375
uses, 392
defined, 370
in DELETE statement, 573–575
as filters, 377–392
predicate keywords, 380–392
sample statements, 400–409
syntax, 378–380, 378*f*
uses, 393–394
row, 370–371
scalar, 370, 372, 373–375
within subqueries, 382–383
subquery UPDATE expressions, 514–516
table, 370, 371–372
types of, 370–372
in UPDATE statement, 508–514
uses for, 392–394
as value expression of aggregate function, 428
WHERE clause of, 373, 380

SUBSTRING, 594*f*
SUM, 416*f,* 421–423, 427–428, 433, 434, 595*f*.
See also aggregate functions
Sybase Enterprise Application Studio, 6, 161
SYMMETRIC BETWEEN predicate, 165
syntax diagrams
IN, 168*f,* 381*f,* 598*f*
aggregate functions, 416*f,* 595*f*
ALL, 487*f,* 598*f*
ANY, 487*f,* 598*f*
asterisk shortcut, 83*f*
AVG, 416*f,* 595*f*
CAST, 110*f,* 594*f*
character string functions, 594*f*
character string literals, 113*f,* 592*f*
column references, 246*f,* 593*f*
comparison predicates, 157*f,* 597*f*
concatenation expression, 118*f*
correlation names for table, 252*f*
COUNT/COUNT(*), 376*f,* 416*f,* 595*f*
date expression, 125*f*
date functions, 594*f*
datetime functions, 594*f*
datetime literals, 115*f,* 592*f*
DELETE, 568*f,* 599*f*
DISTINCT, 85*f*
EXCEPT, 238*f*
EXISTS, 598*f*
filtering with comparison predicate, 378*f*
GROUP BY, 445*f*
HAVING, 476*f*
INSERT
using SELECT expression, 544*f,* 545*f,* 599*f*
with VALUES, 539*f,* 599*f*
INTERSECT, 236*f*
interval literals, 593*f*
IS NULL, 174*f,* 598*f*
JOIN, 596*f*
FULL OUTER, 314*f*
INNER
of three or more tables, 257*f,* 260*f,* 347*f*
of two tables, 247*f,* 257*f,* 305*f*
of more than two tables in alternating sequence, 260*f,* 308*f*
OUTER
of three or more tables, 306*f,* 308*f*
of two tables, 296*f,* 305*f*
using SELECT statements, 302*f*
of three or more tables, 257*f,* 260*f,* 306*f,* 308*f,* 347*f*
of two tables, 247*f,* 257*f,* 296*f,* 305*f*
UNION, 318*f*
LIKE predicate, 169*f,* 598*f*
literals, 113*f,* 114*f,* 115*f,* 592*f*
mathematical expressions, 121*f*
MAX, 376*f,* 416*f,* 595*f*
MIN, 416*f,* 595*f*
naming of expression, 130*f*
NOT, 176*f,* 185*f*
numeric functions, 595*f*
numeric literals, 114*f,* 592*f*
ORDER BY, 351*f*
BETWEEN predicate, 164*f,* 597*f*
quantified predicate, 387*f*
search conditions, 178*f,* 185*f,* 597*f*
SELECT expression, 373*f,* 591*f*
SELECT query, 87*f,* 591*f*
SELECT statement, 74*f,* 78*f,* 81*f,* 135*f,* 372*f*
with all clauses, 476*f*
embedded in INNER JOIN, 255*f*
OUTER JOIN using, 302*f*
UNION of two, 342*f*
SOME, 487*f,* 598*f*
subquery with IN predicate, 381*f*
SUM, 416*f,* 595*f*
time expressions, 126*f*
time functions, 594*f*
TIMESTAMP functions, 594*f*
UNION, 241*f,* 340*f*
ORDER BY clause, 351*f*
of three tables, 349*f*
of two SELECT statements, 342*f*
UPDATE, 502*f,* 599*f*
value expression, 134*f,* 373*f,* 592*f*
WHERE clause, 153*f*
System R, 5, 54–55
Systems Application Architecture (SAA), 61

T
tables
base, defined, 9
correlation names (aliases), 252–254, 252*f*
derived, in INNER JOIN, 254–256, 255*f*

tables (*cont.*)
fine-tuning of, 30–42
INNER JOIN of, 247–251, 247*f*
linking
advantages of, 14–15
defined, 13
defining of, 13–14
in resolving duplicate fields, 37–38, 37*f,* 38*f*
in resolving multivalued fields, 28, 29*f*
logical, defined, 244
naming of, 22, 30–32
overview of, 6–7, 6*f*
primary, defined, 11
relationships between. *See* relationships between tables
secondary, defined, 11
structure, fine-tuning of, 32–33
table subqueries, 370, 371–372
TEXT data type, 107
TIME data type, 109
time expressions, 126–127, 126*f*
time functions
in specific software, 607–614
syntax, 594*f*
TIME keyword, 116, 117
time literals
specifying, 116
syntax, 115*f,* 592*f*
TIMESTAMP data type, 109
TIMESTAMP functions
in specific software, 607–614
syntax, 594*f*
TIMESTAMP keyword, 116
timestamp literals
specifying, 116
syntax, 115*f,* 592*f*
TINYINT data type, 108, 109
TOP, 92
totals. *See* SUM
transactions, 506–507, 554, 570
triggers, 42, 516
TRIM, 594*f*
true, value for, 507–508
tuple, 6. *See also* records
type of participation, setting, 46–47, 47*f,* 48*f*

U
UNION
alternatives to, 352–353, 357–358
column names in, 345
commercial systems' support, 241
of complex SELECT statements, 345–348, 347*f*
CORRESPONDING clause, 343
data type computability, 341–342
multiple, 349–350, 349*f*
overview, 39–342
sample statements, 353–365
set identifier columns and, 348, 357
set requirements, 341
of simple SELECT statements, 342–345, 342*f*
sorting of, 351–352, 351*f*
sample statements, 354–355, 362–363
syntax, 239–241, 241*f,* 340–341, 340*f*
uses of, 215, 232, 233, 352–353
UNION ALL, 228–230, 341, 343, 345, 348
UNION JOIN, 317, 318*f*
union (set operation), 228–233
combining result sets with, 230–232
concept, 228–230
limitations of, 233
set diagrams, 231–232, 232*f*
uses of, 215, 232–233
University of California, Berkeley, 5, 56
UNIX, SQL standards for, 61
unresolved relationships, defined, 13, 13*f*
UPDATE
DISTINCT keyword and, 86
JOIN in, 508–509, 511–512
multiple columns, 507–508
overview, 501–502
sample statements, 503–504, 517–533
selected rows, 504
subquery UPDATE expressions, 514–516
syntax, 502–503, 502*f,* 599*f*
transactions and, 506–507, 554
transforming SELECT statement into, 505, 506*f*
translation process, 503
uses for, 516–517
verifying accuracy of target materials, 505

WHERE clause, 502, 502*f,* 504–505, 506*f,* 507–508
subquery filters, 508–514
USE, 538
USING clause
alternatives to, 301
INNER JOIN, 247, 247*f,* 250–251
OUTER JOIN, 296*f,* 297, 300–301

V
value expressions, 133–135, 134*f*
components of, 134
defined, 134
in SELECT clause, 135, 135*f,* 372–373, 373*f*
sample statements, 139–147
syntax for, 134, 134*f,* 373, 373*f,* 592*f*
VALUES, INSERT statements with, 539–544
sample statements, 552–554, 556, 559–560
syntax, 539*f,* 599*f*
VARCHAR data type, 107
Venn, John, 219
Venn diagrams, 219. *See also* set diagrams
views
defined, 73, 92
overview of, 9–10, 11*f*
Visual Studio, 6

W
"what if?" questions, 106
WHERE clause, 151–156
in aggregate functions, 419, 430
alternatives to, 352–353, 357–358
composition of, 154–156
in DELETE statement, 568, 568*f,* 571–575
sample statements, 576–583
subqueries, 573–575
in multiple UNIONs, 350
predicates (filters) in (*See also* BETWEEN; comparison predicates; IN predicate; IS NULL predicate; LIKE)
efficiency of, 190–191
expressing in different ways, 197–198
INTERSECT, 234–236, 236*f*
multiple conditions, 178–197
NOT operator, 175–178, 176*f,* 184–187, 185*f*
nulls, evaluation of, 193–197, 194*f,* 195*f,* 196*f*
AND operator, 179, 180*f,* 183–184, 195–196, 204–205
order of precedence. *See* order of precedence
OR operator, 180–182, 181*f,* 184–185, 196, 196*t,* 203
AND and OR together, 183–184
overview, 152–154
sample statements, 198–206
in UPDATE statements, 508–514
in SELECT statement, 74*f,* 75
of subquery, 373, 380
syntax of, 152, 153*f*
in UPDATE statement, 502, 502*f,* 504–505, 506*f,* 507–508
subquery filters, 508–514
uses of
generally, 151–152
vs. HAVING, 478–485
Wong, Eugene, 56

X
X3, 57, 62
"X3.135-1992 Database Language SQL," 59–60
X3H2, 57
X/OPEN standard, 61

This page intentionally left blank